

软件开发与测试丛书

# 软件测试
# 实用方法与技术

北 京 跟 踪 与 通 信 技 术 研 究 所

刘文红　张卫祥　司倩然　齐玉华　陈青　马贤颖　杜会森　编著

清华大学出版社

软件开发与测试丛书

# 软件测试实用方法与技术

刘文红　张卫祥　司倩然　齐玉华
陈　青　马贤颖　杜会森　编著

清华大学出版社
北京

## 内 容 简 介

本书全面系统地介绍了软件测试的方法与技术。书中结合实例，详细介绍了动态测试和静态测试中的典型技术方法，比较了各种方法的不同之处并分析了它们的优缺点；紧扣软件测试实际和标准规范要求，从测试原则、测试环境、测试策略、测试内容、测试方法、测试过程等不同方面分别介绍了单元测试、集成测试、配置项测试和系统测试等不同测试级别中实用的测试方法与技术；此外还介绍了常用的软件测试工具，软件测试文档的编写，以及回归测试、面向对象软件测试、FPGA 测试等专门测试。

本书定位于一本软件测试方法和技术的实用指南，适用于软件从业人员了解软件测试的基础知识、一般流程、实用技术方法和常用测试工具，帮助软件从业人员提高技术能力和过程能力水平，也适用于软件测试机构建立测试能力体系，规范软件测试管理。



**图书在版编目(CIP)数据**

软件测试实用方法与技术/刘文红等编著. —北京：清华大学出版社，2017
(软件开发与测试丛书)
ISBN 978-7-302-48066-2

Ⅰ. ①软…　Ⅱ. ①刘…　Ⅲ. ①软件—测试　Ⅳ. ①TP311.55

中国版本图书馆 CIP 数据核字(2017)第 207759 号

**责任编辑**：石　磊
**封面设计**：常雪影
**责任校对**：赵丽敏
**责任印制**：宋　林

**出版发行**：清华大学出版社
**网　　址**：http://www.tup.com.cn，http://www.wqbook.com
**地　　址**：北京清华大学学研大厦 A 座　　**邮　　编**：100084
**社 总 机**：010-62770175　　**邮　　购**：010-62786544
**投稿与读者服务**：010-62776969，c-service@tup.tsinghua.edu.cn
**质量反馈**：010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者**：三河市金元印装有限公司
**经　　销**：全国新华书店
**开　　本**：185mm×260mm　　**印　　张**：19.75　　**字　　数**：480 千字
**版　　次**：2017 年 8 月第 1 版　　**印　　次**：2017 年 8 月第 1 次印刷
**印　　数**：1～3000
**定　　价**：69.00 元

产品编号：072908-01

# “软件开发与测试丛书”
# 编审委员会

# “软件开发与测试”丛书序

为应对“软件危机”的挑战，人们在 20 世纪 60 年代末提出借鉴传统行业在质量管理方面的经验，用工程化的思想来管理软件，以提高复杂软件系统的质量和开发效率，即软件工程化。40 多年以来，软件已广泛应用到各个工程领域乃至生活的各个方面，极大地提高了社会信息化水平，软件工程也早已深入人心。

质量是产品的生命，对软件尤其如此。软件的直观性远不及硬件，软件的质量管理相对困难得多；但与传统行业类似，大型复杂软件的质量在很大程度上取决于软件过程质量。质量评估是质量管理的关键，没有科学的评估标准和方法，就无从有效地管理质量，软件评测是质量评估的最有效和最重要的手段之一。

北京跟踪与通信技术研究所软件评测中心是从事软件评测与工程化管理的专业机构，是在我国大力发展航天事业的背景下，为保障载人航天工程软件质量，经原国防科工委批准，国内最早成立的第三方软件评测与工程化管理的技术实体组织之一。自成立以来，软件评测中心出色地完成了以载人航天工程、探月工程为代表的数百项重大工程关键软件评测项目，自主研发了测试仿真软件系统、测试辅助设计工具、评测项目与过程管理软件等一系列软件测试工具，为主制订了 GB/T 15532—2008《计算机软件测试规范》、GB/T 9386—2008《计算机软件测试文档编制规范》、GJB 141《军用软件测试指南》等软件测试标准，深入研究了软件测试自动化、缺陷分析与预测、可信性分析与评估、测试用例复用等软件测试技术，在嵌入式软件、非嵌入式软件和可编程逻辑器件软件等不同类型软件测试领域，积累了丰富的测试经验和强大的技术实力。

为进一步促进技术积累和对外交流，北京跟踪与通信技术研究所组织编写了本套丛书。本丛书是软件评测中心多年来技术经验的结晶，致力于以资深软件从业者和工程一线技术人员的视角，融会贯通软件工程特别是软件测试、质量评估与过程管理等领域相关的知识、技术和方法。本丛书的特色是重点突出、实用性强，每本书针对不同方向，着重介绍实践中常用的、好用的技术内容，并配以相应的范例、模板、算法或工具，具有很高的参考价值。

本丛书将为具有一定知识基础和工作经验、想要实现快速进阶的从业者提供一套内容丰富的实践指南。对于要对工作经验较少的初入职人员进行技术培训、快速提高其动手能力的单位或机构，本丛书也是一套难得的参考资料。

**丛书编审委员会**

2015 年 5 月 6 日

# 前　言

软件测试是保障软件质量的重要手段，是构建高可信软件的关键环节。随着软件应用的日益广泛，人们对软件测试重要性的认识越来越深刻。20 世纪 80 年代以来，特别是在最近一二十年间，软件测试无论是作为一项技术、一门学科还是作为一个行业，都得到了快速蓬勃的发展。

本书定位于一本软件测试方法和技术的实用指南，紧扣软件测试实际和标准规范要求，结合行业内软件测试现状，系统地介绍软件测试相关的知识、方法、技术和软件工具，给出较为详细的软件测试过程技术文档模板。

本书旨在帮助软件从业人员了解软件测试的基础知识、一般流程、实用技术方法和常用测试工具，提高技术能力和过程能力水平，以及帮助软件测试机构建立测试能力体系，规范软件测试过程管理。

本书是作者多年从事软件测试工程实践和技术方法研究的经验总结，与其他公开教材相比，主要特色有：实用性强，本书紧扣软件测试实际和标准规范要求，着重介绍测试实践中常用和好用的知识、技术和方法；视角特殊，本书从第三方测评的角度，系统地阐述适用于工程实践的测试方法和技术，反映行业实际需求和技术发展动态。

本书共有 12 章，可分为 5 大部分。

第一部分(第 1 章)是软件测试概述，简要介绍软件测试发展历程、软件测试典型定义、软件测试一般原则、常用软件测试模型、常见软件测试级别与测试类型、软件测试相关的标准规范等基本内容。

第二部分(第 2、3 章)是软件测试技术，结合程序实例，分别介绍静态测试技术和动态测试技术，并对典型技术进行比较分析。

第三部分(第 4～7 章)按照不同的测试级别，从测试原则、测试环境、测试策略、测试内容、测试方法、测试过程等方面分别介绍单元测试、集成测试、配置项测试和系统测试中的实用测试方法与技术。

第四部分(第 8～10 章)以独立章节分别介绍回归测试、面向对象软件测试和 FPGA 测试等专门测试类别中的实用测试方法与技术。

第五部分(第 11、12 章)主要内容是测试实践，第 11 章分别介绍了在静态测试、动态测试和测试管理中常用的软件工具。第 12 章着重介绍测试策划、测试设计、测试实施和测试总结等测试过程中常用的技术文档，给出通用技术文档模板。

软件测试包含大量相关的活动，有些是技术性的，有些是管理性的，还有些是相互交织的。例如，单元测试、集成测试、配置项测试、系统测试、回归测试中测试用例、测试数据和测试期望结果的设计是典型的技术性活动；人员计划、成本预算以及配置管理、项目监控中的大部分内容是典型的管理性活动。如前所述，本书偏重于讲述软件测试的技术性活动，软件

测试的管理性活动将在本套丛书的另一本书《软件测试管理》中重点介绍。

本书第 1～3 章由张卫祥编写，第 4～7 章和第 12 章由刘文红编写，第 8 章由陈青编写，第 9 章由齐玉华编写，第 10 章由司倩然编写，第 11 章由马贤颖和司倩然编写。全书由刘文红、杜会森统稿。赵辉、张卫民、杨宝明、李国华、牛胜芬等专家审阅了初稿并提出了许多宝贵意见。

在本书编写过程中，得到了北京跟踪与通信技术研究所，特别是软件评测中心的大力支持，还得到了编者同事、朋友和家人的关心与帮助，在此一并表示感谢！

由于水平有限，本书肯定还存在不少问题，敬请大家批评指正。

**编　者**
2017 年 4 月

# 目　录


**第 1 章　软件测试概述** …… 1

1.1　软件测试简史 …… 1

1.2　软件测试定义 …… 3

1.3　软件测试原则 …… 4

1.4　软件的可测试性 …… 6

1.4.1　可测试性定义与内涵 …… 6

1.4.2　可测试性设计与实现 …… 7

1.4.3　可测试性度量与评估 …… 9

1.5　软件测试模型 …… 9

1.5.1　V 模型 …… 9

1.5.2　W 模型 …… 10

1.5.3　H 模型 …… 11

1.6　软件测试级别与测试类型 …… 12

1.6.1　软件测试级别 …… 12

1.6.2　软件测试类型 …… 13

1.6.3　软件关键等级 …… 16

1.7　软件测试标准规范 …… 18

1.7.1　相关标准概述 …… 18

1.7.2　GB/T 9386—2008《计算机软件测试文档编制规范》 …… 21

1.7.3　GB/T 15532—2008《计算机软件测试规范》 …… 21

1.7.4　GB/T 25000.51—2010《软件工程软件产品质量要求和评价(SQuaRE)商业现货(COTS)软件产品的质量要求和测试细则》 …… 21

1.7.5　ISO/IEC 29119 *Software Testing* …… 22

1.8　软件测试人员能力素质要求 …… 26

1.9　术语与缩略语 …… 27

**第 2 章　静态测试技术** …… 29

2.1　文档审查 …… 29

2.1.1　实施要点 …… 30

2.1.2　组织与流程 …… 30

2.1.3　成果形式 …… 31

2.2 代码审查 …… 34
2.2.1 实施要点 …… 34
2.2.2 组织与流程 …… 35
2.2.3 成果形式 …… 35
2.3 静态分析 …… 38
2.3.1 实施要点 …… 38
2.3.2 组织与流程 …… 39
2.3.3 成果形式 …… 40
2.4 代码走查 …… 40
2.4.1 实施要点 …… 40
2.4.2 组织与流程 …… 40
2.4.3 成果形式 …… 41
2.5 静态测试技术分析 …… 41

**第3章 动态测试技术 …… 43**
3.1 白盒测试 …… 43
3.1.1 概述 …… 43
3.1.2 白盒测试基础 …… 46
3.1.3 基本路径测试 …… 51
3.1.4 控制结构测试 …… 53
3.1.5 其他白盒测试技术 …… 61
3.2 黑盒测试 …… 64
3.2.1 概述 …… 64
3.2.2 等价类划分 …… 65
3.2.3 边界值分析 …… 68
3.2.4 因果图与决策表法 …… 70
3.2.5 组合测试 …… 75
3.2.6 基于场景测试 …… 77
3.2.7 错误推测法 …… 81
3.2.8 黑盒测试技术分析 …… 84
3.3 灰盒测试 …… 85
3.3.1 概述 …… 85
3.3.2 实施步骤 …… 86
3.3.3 灰盒测试技术分析 …… 86
3.4 动态测试技术分析 …… 87

**第4章 单元测试 …… 89**
4.1 概述 …… 89
4.1.1 单元测试的定义 …… 89

4.1.2 单元测试的目的 …… 89
4.1.3 单元测试的重要性 …… 90
4.2 单元测试原则 …… 92
4.3 单元测试环境 …… 92
4.4 单元测试策略 …… 94
4.4.1 自顶向下 …… 94
4.4.2 自底向上 …… 94
4.4.3 独立单元 …… 95
4.5 单元测试内容 …… 95
4.5.1 功能测试 …… 96
4.5.2 性能测试 …… 96
4.5.3 接口测试 …… 96
4.5.4 局部数据结构测试 …… 96
4.5.5 边界条件测试 …… 97
4.5.6 独立执行路径测试 …… 97
4.5.7 错误处理测试 …… 97
4.6 单元测试方法 …… 98
4.6.1 静态测试 …… 98
4.6.2 动态测试 …… 99
4.7 单元测试用例设计 …… 100
4.8 单元测试过程 …… 101
4.8.1 测试策划 …… 102
4.8.2 静态测试 …… 103
4.8.3 动态测试 …… 111
4.8.4 测试总结 …… 113

**第5章 集成测试 …… 114**
5.1 概述 …… 114
5.1.1 集成测试的定义 …… 114
5.1.2 集成测试的目的 …… 115
5.1.3 集成测试的重要性 …… 116
5.2 集成测试原则 …… 117
5.3 集成测试环境 …… 117
5.4 集成测试策略 …… 118
5.4.1 大爆炸式集成 …… 119
5.4.2 自顶向下集成 …… 120
5.4.3 自底向上集成 …… 122
5.4.4 三明治式集成 …… 123
5.4.5 核心系统先行集成 …… 124

5.4.6 分层集成 125
5.4.7 基于功能的集成 126
5.4.8 高频集成 127
5.4.9 基于进度的集成 128
5.4.10 基于使用的集成 128
5.4.11 基于风险的集成 129
5.4.12 客户/服务器系统的集成 129
5.5 集成测试内容 130
5.6 测试方法 131
5.6.1 体系结构分析 131
5.6.2 模块分析 131
5.6.3 接口分析 132
5.6.4 可测试性分析 133
5.6.5 集成测试策略分析 133
5.7 集成测试用例设计 133
5.8 集成测试过程 134
5.8.1 测试策划 135
5.8.2 测试设计与实现 137
5.8.3 测试执行 138
5.8.4 测试总结 138

**第 6 章 配置项测试 140**
6.1 概述 140
6.1.1 配置项测试的定义 140
6.1.2 配置项测试的目的 140
6.1.3 配置项测试的重要性 141
6.2 配置项测试原则 141
6.3 配置项测试环境 142
6.4 配置项测试策略 143
6.5 配置项测试内容 143
6.6 配置项测试方法 144
6.6.1 功能测试 145
6.6.2 性能测试 146
6.6.3 接口测试 147
6.6.4 人机交互界面测试 148
6.6.5 强度测试 149
6.6.6 余量测试 149
6.6.7 安全性测试 150
6.6.8 恢复性测试 151

6.6.9 边界测试 152
6.6.10 数据处理测试 152
6.6.11 安装性测试 153
6.6.12 容量测试 154
6.7 配置项测试用例设计 154
6.7.1 概述 154
6.7.2 SFME&FTA 综合分析 155
6.7.3 建立软件测试用例设计模式 158
6.7.4 应用实例 159
6.8 配置项测试过程 161
6.8.1 测试策划 162
6.8.2 测试设计与实现 163
6.8.3 测试执行 164
6.8.4 测试总结 164

**第 7 章 系统测试 166**
7.1 概述 166
7.1.1 系统测试的定义 166
7.1.2 系统测试的目的 166
7.1.3 系统测试的重要性 167
7.2 系统测试原则 167
7.3 系统测试环境 168
7.4 系统测试策略 168
7.5 系统测试内容 169
7.6 系统测试方法 169
7.6.1 可靠性测试 169
7.6.2 互操作性测试 172
7.6.3 兼容性测试 173
7.7 系统测试用例设计 174
7.7.1 概述 174
7.7.2 系统形式化模型 175
7.7.3 基于模型的系统测试 178
7.7.4 实例 182
7.8 系统测试过程 184

**第 8 章 回归测试 185**
8.1 概述 185
8.1.1 回归测试的定义 185
8.1.2 回归测试的目的 186

8.1.3 回归测试的重要性 …… 186
8.2 回归测试策略 …… 187
8.3 软件更动影响域分析方法 …… 187
8.3.1 黑盒测试更动影响域分析 …… 187
8.3.2 白盒测试更动影响域分析 …… 191
8.4 回归测试用例设计 …… 195
8.4.1 回归测试用例设计原则 …… 195
8.4.2 已有测试用例的选取 …… 195
8.5 回归测试过程 …… 198

**第9章 面向对象软件测试 …… 200**
9.1 面向对象软件简介 …… 200
9.2 面向对象软件测试概述 …… 203
9.2.1 面向对象软件的特点对测试的影响 …… 203
9.2.2 面向对象软件测试和传统测试的不同 …… 204
9.2.3 面向对象软件测试分类 …… 205
9.3 面向对象软件测试模型 …… 205
9.3.1 面向对象分析测试 …… 206
9.3.2 面向对象设计测试 …… 208
9.3.3 面向对象编程测试 …… 208
9.3.4 面向对象单元测试 …… 209
9.3.5 面向对象集成测试 …… 212
9.3.6 面向对象系统测试 …… 215

**第10章 FPGA测试 …… 218**
10.1 FPGA测试概述 …… 218
10.1.1 可编程逻辑器件的基本概念 …… 218
10.1.2 硬件描述语言的发展历程 …… 219
10.1.3 VHDL语言 …… 219
10.1.4 Verilog HDL语言 …… 220
10.1.5 面向可编程逻辑器件的开发过程 …… 220
10.1.6 可编程逻辑器件软件与传统软件的不同 …… 222
10.1.7 全过程域的可编程逻辑器件测试框架 …… 223
10.2 静态测试 …… 223
10.2.1 文档审查 …… 224
10.2.2 代码审查 …… 227
10.2.3 编码规则检查 …… 229
10.2.4 跨时钟域分析 …… 230
10.2.5 等效性验证 …… 235

10.2.6 静态时序分析…… 239

10.3 仿真测试…… 243

10.3.1 仿真测试的特点…… 244

10.3.2 仿真测试平台的组成…… 245

10.3.3 仿真测试的流程…… 245

10.3.4 功能仿真测试…… 247

10.3.5 门级仿真测试…… 248

10.3.6 时序仿真测试…… 248

10.3.7 仿真测试支持工具…… 249

10.4 软硬协同验证…… 250

10.4.1 验证环境构成…… 250

10.4.2 支持工具…… 251

10.5 板级验证…… 251

10.5.1 作用…… 251

10.5.2 板级验证的典型环境…… 252

10.5.3 板级验证的流程…… 252

**第11章 测试工具 …… 254**

11.1 概述…… 254

11.2 静态测试工具…… 255

11.2.1 Logiscope …… 255

11.2.2 PRQA …… 257

11.2.3 SpyGlass …… 259

11.2.4 PrimeTime …… 261

11.2.5 Formalpro …… 261

11.2.6 其他静态测试工具…… 262

11.3 动态测试工具…… 262

11.3.1 QACenter …… 262

11.3.2 WinRunner …… 265

11.3.3 JUnit …… 266

11.3.4 Testbed …… 268

11.3.5 CodeTest …… 270

11.3.6 QuestaSim …… 271

11.3.7 其他动态测试工具…… 272

11.4 测试管理工具…… 272

11.4.1 TestCenter …… 272

11.4.2 TP-Manager …… 274

11.4.3 其他测试工具…… 278

**第 12 章　软件测试文档** …… 279
12.1　概述 …… 279
12.2　制定测试计划 …… 280
12.2.1　测试计划内容 …… 280
12.2.2　测试计划模板 …… 281
12.2.3　测试计划常见问题 …… 284
12.3　测试设计与实现 …… 285
12.3.1　测试设计与实现的内容 …… 286
12.3.2　测试说明模板 …… 287
12.3.3　测试设计与实现常见问题 …… 288
12.4　测试执行 …… 290
12.4.1　测试执行的内容 …… 290
12.4.2　测试执行模板 …… 290
12.4.3　测试实施常见问题 …… 292
12.5　测试总结 …… 293
12.5.1　测试总结的内容 …… 293
12.5.2　测试总结模板 …… 294
12.5.3　测试总结常见问题 …… 296

**参考文献** …… 298

# 软件测试概述

随着科学技术的迅速发展，人类已经进入信息社会。信息的获取、处理、交流和决策都需要大量的软件，软件成为了人们工作和生活中不可或缺的工具。软件的应用日益广泛，软件的质量受到人们越来越多的关注。特别是在航空航天、金融保险、交通通信、工业控制等重要领域，软件一旦失效将造成重大损失，因此对软件质量提出了更高的要求。

1996年6月，阿丽亚娜5(ARIANE 5)火箭在历时十年研制后的首次发射中，由于软件故障造成火箭升空40s后即发生爆炸，直接经济损失达到5亿美元，还使得耗资80亿美元的开发计划推迟了近3年。2003年5月，俄罗斯TMA1号宇宙飞船，由于软件错误导致导航系统故障，造成飞船从国际空间站返回地面时与飞行控制中心失去联系长达11min，飞船最终降落在与预定降落点偏差460多千米的地方。2003年8月，由于FirstEnergy公司电力监测与控制管理系统的预警服务出现软件错误，导致多个重要设备出现故障，而操作员在一个小时后才得到控制站的指示，造成美国及加拿大部分地区史上最大停电事故。

软件测试是保障软件质量的重要手段，是构建高可信软件的关键环节。随着人们对软件测试重要性的认识越来越深刻，软件测试阶段在整个软件开发周期中所占的比重日益增大。统计数据表明，软件测试占软件开发总成本的比例一般达到50%以上。现在有些软件开发机构将40%以上的研制力量投入到软件测试之中；对于某些性命攸关的软件，其测试费用甚至高达所有其他软件工程阶段费用总和的3～5倍。尽管人们在软件开发过程中也采用形式化方法描述和证明软件规约，并采用程序正确性证明、模型检验等方法保证软件质量，但是这些方法都存在一定的局限性，尚未达到广泛实用阶段。软件测试在今后较长时间内仍将是保证软件质量的重要手段。

## 1.1　软件测试简史

人们对软件测试的认识是逐步发展起来的，如图1-1所示。在20世纪50年代，计算机技术发展初期，软件规模都很小，复杂度相对较低，软件错误大部分在开发人员的调试阶段就发现解决了，软件测试被定义为“程序员为了在他们的程序中找到bug所做的事情”。在这个阶段，测试和调试是等同的，都是由开发人员自己完成。

在20世纪60年代早期，软件测试与调试区分开来，测试主要在程序编写后进行。人们开始考虑以遍历代码的可能路径或可能的输入变量的方式，对软件进行“彻底”的测试。现

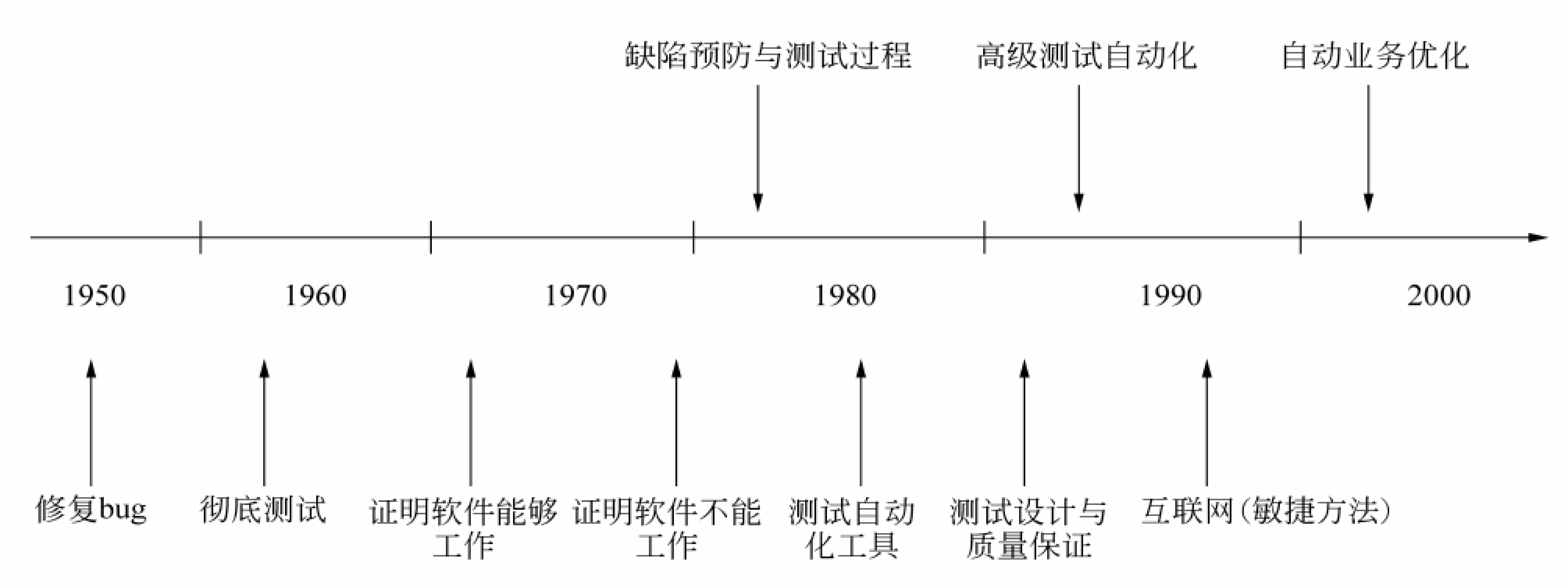



**图 1-1 软件测试的发展**

在我们知道,由于程序的输入域太大、有太多可能的输入路径以及设计和规约等问题很难测试,完全彻底地测试一个程序是不可能的。

在 20 世纪 70 年代早期,随着软件开发的成熟,人们开始提出软件开发的工程化思想,软件测试得到发展,软件测试的地位得到确认。软件测试被定义为"证明一个程序的正确性而要做的事情",即证明软件能够工作。这期间提出的"正确性证明技术",建议在软件分析、设计和实现过程中通过证明的方式进行软件正确性验证,在理论上很有前途,但在实践中过于耗时、效率极低。

在 20 世纪 70 年代后期,测试被定义为带着找到错误的意图执行程序的过程,而不是证明程序能够运行的过程。与上一观点相反,这种观点强调了好的测试用例能够有更大几率去发现尚未发现的错误,成功的测试是发现了尚未发现的错误的测试。

20 世纪 80 年代,随着软件行业的飞速发展,软件测试的理论和技术得到了快速的发展,人们开始把软件测试作为软件质量保证的重要手段,认识到软件测试不是一次性在开发后期完成的,软件测试也不再仅限于程序本身,软件测试的定义被扩展到缺陷预防,软件测试活动被扩充到对需求、设计、编码、测试等整个软件开发生存周期的过程控制中。IEEE 对软件测试定义为"使用人工或自动的手段来运行或测定某个软件系统的过程,其目的在于检验它是否满足规定的需求或弄清楚预期结果与实际结果之间的差别",并发布了软件测试文档的国际标准。

在 20 世纪 80 年代中期,自动化测试工具出现了。相比于手动测试,自动化测试工具的引入大幅提高了测试的效率和质量。测试工具起初还是相当原始的,不具有高级脚本语言工具。

在 20 世纪 90 年代早期,从质量保证的观点,测试的含义被定义为"策划、设计、建造、维护和执行测试及测试环境",软件测试的过程应是受管理的,其自身也存在一个生存周期。更多的录制/回放测试工具提供了丰富的脚本语言和报告功能,测试管理工具可帮助管理从测试需求分析和设计到测试脚本和缺陷的所有内容,也有一些商用的性能工具来测试系统的性能,能够进行压力和负载测试等。

在 20 世纪 90 年代中期,人们仍认为测试应是一个贯穿整个软件开发生存周期的过程,但是随着互联网的流行,软件的生存周期模型和开发模式发生了很大的变化,使得软件测试变得越来越困难。测试有时在没有明确预先定义所有测试方向的情况下进行,这

种测试方法被认为是敏捷测试，其他测试技术还有探索测试、快速测试和基于风险的测试等。

随着信息科学和软件开发技术的发展，人们对软件测试的概念及其作用的认识已趋于成熟和稳定，进入 21 世纪以来，对软件测试的研究主要集中于软件测试的技术方法和最佳实践上，目的是提升软件测试的效率和效果。自动业务优化(BTO)的基本思想是衡量软件在其整个生存周期中的价值并使其最大化，以确保软件达到可用性、性能、质量和经济相协调的目标。软件测试作为软件整个生存周期的一项重要活动被内置其中。

由于人们对软件质量越来越关注，对软件测试越来越重视，可以预测，在未来很长的时间内，软件测试将会得到快速发展，软件测试理论将更加完备，软件测试工具将更加丰富，软件测试行业将蓬勃发展，测试人员素质和能力将不断提升。测试有效性和效率最大化是软件测试的共性问题，测试自动化和智能化将成为软件测试的重要发展方向之一。

## 1.2　软件测试定义

在 GB/T 11457—2006《信息技术　软件工程术语》中，测试被定义为“a)在规定的条件下操作系统或部件、观察或记录结果并对系统或部件的某些方面作评价的过程；b)分析软件项以检测在存在的和要求的条件之间的区别(即，隐错)以评价软件项的特征”。也就是说，软件测试可分为确认和验证两部分。

“确认”指的是检查所开发的软件是否满足用户真正需求的活动。从用户的真正需要出发，对软件需求和设计说明存疑，以发现软件需求定义、产品设计和程序实现中的问题。

“验证”指的是检验软件是否正确实现了软件需求规格书、产品设计说明等文档中所定义的功能和特性的活动。验证过程应提供证据以表明软件产品与研制要求相一致，或表明两者之间的差异。

人们对软件测试的认识是不断深入的。随着软件测试的持续发展，曾出现了许多对软件测试的重要观点或定义。这些观点一度非常流行，有些至今仍很有价值，表现了人们从特定角度对软件测试的认识。

“(软件测试)就是建立一种信心，表明程序能够按照预期的设想运行”，这是软件测试先驱 Bill Hetzel 在 1973 年给出的软件测试的定义。后来，Bill Hetzel 在其 1983 年出版的经典著作 *The Complete Guide to Software Testing* 中，对上述定义进行了修订：“测试是评价一个程序或者系统的属性和能力的各种活动，并确定它是否达到预期的结果。”Bill Hetzel 的核心观点是：测试的目的是验证软件是“工作的”，即以正向思维方式，针对软件系统的所有功能点，逐个验证其正确性。这种观点比较符合用户的角度。对于用户来说，适当的测试是能够接受软件或系统的依据。用户通过亲自或委托第三方的测试，获得测试结果，对软件系统的质量进行判断并做出最后的抉择。

“测试是为了发现错误而执行程序的过程”，这是另一位软件测试先驱 Glenford Myers 1979 年在其著名的著作 *The Art of Software Testing* 中给出的定义。后来，Glenford Myers 对该定义进行了完善，进一步提出了他对软件测试的重要观点：测试是为了证明程序有错，而不是证明程序无错误；一个好的测试用例在于它能发现至今未发现的错误；一个

成功的测试是发现了至今未发现的错误的测试。这种观点的核心是证明软件是“不工作的”,即以反向思维,不断思考软件或系统的弱点,发现软件或系统中存在的问题。这种观点比较符合软件技术人员的角度。对开发方来讲,软件测试可尽早地发现和改正软件中的错误,避免给用户造成损失和给开发方的信誉造成不良影响。

“(测试是)使用人工或自动的手段来运行或测定某个软件系统的过程,其目的是检验软件系统是否满足规定的需求或弄清预期结果与实际结果之间的差别”,这是 1983 年 IEEE 软件工程术语标准中对软件测试给出的定义。这个定义明确指出软件测试的目的是检验软件系统是否满足需求。软件测试不再是一个一次性的、软件开发后期的活动,而是与整个开发流程融为一体。现在我们认为,软件测试是贯穿整个软件开发生存周期、对软件产品(包括阶段性产品)进行验证和确认的活动过程,其目的是尽早、尽快地发现软件产品中存在的各种问题以及与用户需求、预先定义不一致的地方。

软件测试还存在如下重要特征:软件测试是不完全的或无法穷尽的;软件测试是证错的而不证对的,即不完全的软件测试无法证明软件的正确性,只能证明软件的不正确性。

软件测试是不完全的或无法穷尽的,是因为如下 3 点:

(1) 测试是几乎不可能 100%覆盖的。一般来说,由于软件的复杂性和大规模,软件的输入空间非常庞大,程序中可以执行的路径无法穷尽。如果有充足的条件进行不断的测试,总是可以找到更多的缺陷。

(2) 测试环境难以和实际运行或用户环境完全吻合。某些缺陷只有在用户环境下才存在,有些缺陷只有在软件运行一段时间后,随着过量或无效数据的积累才会发生。

(3) 测试人员对产品的理解不能完全代表实际用户的理解。这两者之间的差异意味着可能会存在某些对测试人员来说不是缺陷但对于用户来说是缺陷的缺陷。

软件测试是证错而不证对的,即不完全的软件测试无法证明软件的正确性,只能证明软件的不正确性。这主要是因为如下 2 点:

(1) 发现的错误越多并不能说明软件中剩余的错误就越少,相反地,发现错误越多的地方往往漏掉错误的可能性越大。软件错误存在“群集现象”,根据“二八原理”,20%的代码可能包含了软件中 80%的错误。

(2) 修正过去的缺陷往往会导致新缺陷的产生。需求总是变化的,软件系统不是一成不变的,变是永恒的,“变”可能会带来新的缺陷。

## 1.3 软件测试原则

软件测试的最终目标是提高客户的满意度,而在交付软件前发现尽可能多的缺陷将有助于达到该目标。好的测试应该能用最少的时间和最小的代价发现不同类型的错误。

软件测试的目标可表述为 5 个方面:验证、确认、缺陷预防、质量改进和可靠性评价。验证是证实软件已满足其技术规格说明的过程,用来检查每个开发活动得到的产品,以确定软件开发的结果是否满足对应的需求;确认过程用来证实软件已满足其业务需求,确认活动可以在软件开发生存周期中实施,以保证所有的需求已被满足;缺陷是指软件在使用过程中其预期结果与实际结果之间存在的差异,开发和维护大规模应用软件所面临的主要挑战是

如何尽早地识别缺陷；质量可被一组称为特性的属性进行描述，这些质量属性可以通过软件测试以进行改进；软件可靠性指的是在规定的时间和环境下，软件无失效运行的概率，可以通过分析测试获得的数据进行可靠性评价。

根据软件测试的目的与定义，软件测试的一般原则包括以下 15 点：

（1）尽早地和不断地进行软件测试。由于软件的抽象性和复杂性，在软件开发的每个阶段都可能引入缺陷。数据显示，软件缺陷发现得越早，修复缺陷的成本就越低；问题发现得越晚，解决问题的代价就越大。因此，从需求分析甚至更早的时候开始，软件开发过程中就应该开展软件测试活动。同时，软件测试应该贯穿整个软件开发生存周期，在软件开发的每个环节不断地进行测试，才能及时地发现错误，避免错误的累积和扩大。目前较新的极限编程和测试驱动开发技术更是强调测试先行，提出在编码开始之前就编写测试用例，用测试指导代码编码。

（2）测试要有停止准则，适可而止。在有限的时间和资源条件下找出所有的软件缺陷，进行完全的测试，几乎是不可能的。测试应尽可能多地发现缺陷，特别是严重等级较高的、影响软件正常运行的缺陷，但无法穷尽。根据实际情况，决定测试范围和测试程度并进行有效地控制，才能更好地协调开发与测试的关系，以最少成本获得最大回报，达到测试的理想效果。

（3）软件测试须具有一定的独立性。程序员应尽可能地避免测试自己编写的程序，开发小组应尽可能地避免测试本小组开发的程序。如果条件允许，应由相对独立的测试部门或第三方测试机构进行测试；或者至少应采取互相自测的方式。这主要是因为程序员不愿意且也不容易发现自己编写的程序中的错误。

（4）软件测试应追溯到用户需求。软件开发的最终目的是满足最终用户的需求，使用户按预期完成指定的任务。软件中最严重的错误是那些导致软件无法满足用户需求的缺陷。软件测试人员应紧紧围绕用户需求，站在用户角度看问题。不满足用户需求的软件是无法顺利交付的，这也是验收测试以用户需求为依据的原因。

（5）应进行有重点的测试。从风险观点来讲，软件测试可以被定义为对软件中潜在的各种风险进行评估的活动。由于不可能进行完全测试，从某种意义上，测试了什么比测试了多少更重要。所以，应进行软件风险分析，根据风险大小确定测试的优先级和执行顺序。

（6）应认真做好测试计划。合理的计划是成功的一半，测试前应与用户、开发方沟通，对被测软件进行分析，明确测试目的、测试范围、测试环境、测试方法以及测试所需的资源等内容，形成测试计划。软件测试应按计划进行，切忌随意性。

（7）设计完善的测试用例。测试用例除了要对输入数据进行详细描述，还应给出预期输出结果，否则，很可能在测试过程中因无法与预期结果比对，错过本应发现的错误。另外，测试用例不仅应包括有效或合法的输入，还要考虑无效或非法的输入情况。程序员往往基于预料之中的输入编写程序，而忽视对非预料输入的处理，使得程序留有隐患。

（8）对测试用例进行管理。应重视测试过程中的记录并保存相关文档，测试用例作为软件测试设计阶段的主要工作成果，是软件测试的最重要内容之一。除了一次性软件，几乎所有的软件测试都不可能一次性完成。在生存周期中，软件一般都要经历多次修改。已有的测试用例可以为再次测试时测试用例的选择性复用、新测试用例的生成等提供良好依据

和基础，避免重复劳动，提高测试效率。

(9) 选择合适的测试方法。无论是动态测试还是静态测试技术，都有很多测试方法。应根据不同软件不同部分的不同特点，选用不同的测试方法，开展充分而有针对性的测试。对测试方法的选择是测试策略的重要内容。

(10) 使用恰当的测试工具。测试工具能够扩展测试人员的能力。测试工作是细致而繁琐的，有许多工作，比如某些测试数据的生成、某些测试结果比对等，如果手工完成，既累人又容易出错。恰当使用测试工具能够提高测试的正确性，并且使测试人员有更多精力去从事创造性的工作，提高测试的效果和效率。

(11) 对实测结果进行确认。测试用例执行结果中会存在大量的被测软件输出正确和错误信息，如果不仔细检查，可能会使错误被漏掉。每个测试用例承担不同的测试任务，必须对用例预期的输出结果进行明确定义，对用例的实测结果进行仔细分析检查和确认。

(12) 对软件缺陷进行分析。缺陷之间往往存在相互关联或依赖关系。事实上，数据表明，软件缺陷存在群集现象，即大部分的软件缺陷存在于小部分的软件代码中。通过分析所发现的缺陷的类型、来源、模式和趋势等，可以进一步挖掘新问题，找到软件需要改进的地方。

(13) 进行回归测试。由于缺陷之间的关联性，程序员在修正缺陷后，有可能会引入其他的缺陷；另外，当需求变更或增加新的需求时，对原有软件也具有波及效应，可能会导致新缺陷的产生。因此，当软件更动时，应进行回归测试，以保证已有缺陷被正确关闭且没有引入新的缺陷，软件中原先能正常运行的部分依然正常工作。回归测试是软件生存周期中的一个重要组成部分。

(14) 不断进行培训。软件测试是一项富有创造性的和挑战性的工作，软件测试同其他学科一样，有着一套特定的知识体系和技能要求。根据测试人员执行测试类型的不同，甚至还需要特殊的或先进的知识或技能。测试人员需要不断地进行培训或自我培训，才能适应测试工作中的新挑战。

(15) 测试即服务。软件测试的标准追溯到用户的需求；软件测试是贯穿整个软件生存周期的活动；软件测试人员通过对软件产品进行研究，获取软件信息提供给用户或项目决策者以帮助其做出正确的决定。把软件测试理解为一种服务，对用户的服务，对软件最终质量的服务，有利于软件测试工作的开展。

## 1.4 软件的可测试性

### 1.4.1 可测试性定义与内涵

不同软件具有的自身特点，使得对其进行软件测试和测试结果分析的难易程度不同，这也反映了不同软件可测试性的高低。可测试性是度量软件是否容易被测试的一种属性，是软件开发与测试中的一个重要概念。具有更高可测试性的软件能够使得测试更加容易，能够降低测试成本和工作量，并且往往使软件具有更好的质量。提高软件可维护性的一种途

径是针对可测试性进行软件设计，通常需要考虑软件规模、复杂度、系统结构、分布性、不确定性等多种因素。

Voas 将软件可测试性定义为“当软件本身存在故障并按一种特定的输入分布被执行时，软件会发生失效的概率”。该定义把可测试性与导致软件失效的能力以及软件运行环境联系起来，还有其他定义把可测试性与软件内部可见性、对软件的控制能力、对失效的检测能力、系统的规模或复杂度等相关联。

Bach 把可测试性定义为一个软件能够被测试的难易程度，其内部要素包括可操作性、可控制性、简单性、稳定性、可理解性和可观察性。

(1) 可操作性：指软件在给定配置和环境下被操作和使用的能力。

(2) 可控制性：指测试人员为了测试软件在极端环境下的运行而创建复杂场景的难易程度。

(3) 简单性：指软件实现规定用途的能力水平，包括功能简单性、结构简单性和代码简单性。功能简单性实现了需求所规定的最少特征，结构简单性使模块之间的交互和耦合易于理解，代码简单性使代码易于检查、评审和走查。

(4) 稳定性：指软件不需要经常变更，如果变更，其变更是可控的，变更结果不会影响自动化测试的效果。

(5) 可理解性：指软件——包括程序和规格说明、接口文档、测试文档、用户手册等产品文档等——是否易于理解。

(6) 可观察性：指利用测试工具从被测软件捕获数据和信息以便测试人员对测试结果进行分析和比对的能力，也包括利用人工方式对隐含信息进行测试的能力。

### 1.4.2　可测试性设计与实现

软件可测试性表征了一个软件在给定测试环境下能够被测试的程度。如果软件缺少可测试性，可能会导致软件错误在软件生存周期后期才被发现并付出更大的修复代价。因此，在软件生存周期早期就应该强调软件可测试性。事实上，如同软件测试一样，可测试性应该贯穿于软件开发生存周期的所有阶段。

在需求阶段，开发和测试人员应该充分理解用户需求，在规格说明中清晰定义软件预期功能以及在不正常场景下软件的预期行为。对软件可测试性，主要包括：

(1) 清晰定义可测试性需求；

(2) 清晰定义可度量的需求；

(3) 提出为更好进行软件测试而定义的需求；

(4) 评审软件需求的可测试性，确保其是可测试的和可度量的。

在设计阶段，针对需求阶段的各种软件需求进行有效而详细的设计。对软件可测试性，主要包括：

(1) 加强软件体系结构模型和接口设计的可测试性；

(2) 定义软件设计模式、标准和框架以提高可测试性；

(3) 评审软件设计的可测试性。

在编码阶段，依据需求阶段和设计阶段的要求，编码以实现软件程序。对软件可测试

性，主要包括：

（1）完成可测试的构件并进行内置测试；

（2）生成软件测试配套工具和可重用框架；

（3）根据定义好的可测试性标准对软件程序进行代码检查。

在测试阶段，对前面阶段的可测试性措施以及软件功能性和非功能性需求进行测试。对软件可测试性，主要包括：

（1）定义可到达的软件测试准则和测试脚本；

（2）针对构件开发，设置和使用测试床和相关工具；

（3）基于定义的准则对测试覆盖率进行监控；

（4）验证、度量和评价软件可测试性。

从其内部要素考虑，可测试性设计和实现主要体现在以下6个方面：

（1）可操作性方面：

① 系统具有较少缺陷；

② 没有缺陷能够阻止测试执行；

③ 产品在功能定义阶段进行演化；

④ 可以得到源代码。

（2）可控制性方面：

① 测试环境可用；

② 测试人员能够直接控制变量；

③ 模块、对象或函数层可以独立测试。

（3）简单性方面：

① 设计本身是一致的；

② 功能简单；

③ 结构简单；

④ 代码简单。

（4）稳定性方面：

① 软件变更不频繁；

② 变更是可控的；

③ 软件变更不会使得自动化测试失效。

（5）可理解性方面：

① 设计与其他已知产品类似；

② 产品所基于的技术被广泛理解；

③ 内部、外部和共享构件之间的依赖性被广泛理解。

（6）可观察性方面：

① 每个输入产生不一样的输出；

② 变量在执行过程中是可见或可查询的；

③ 影响输出的所有因素是可见的，不正确的输出很容易被识别；

④ 能够检测到内部错误。

### 1.4.3 可测试性度量与评估

一般地，当我们说一个软件相比于另一个软件具有更好的可测试性，意味着该软件更容易被测试。可采用多种方法来度量软件可测试性。

白盒测试和黑盒测试是常用的 2 种不同的测试策略。黑盒测试把被测软件看做一个黑盒子，依据规格说明设计测试用例、注入软件输入，捕获软件输出并与期望结果相比对，以判断被测软件的正确性；白盒测试把被测软件看做一个透明的盒子，在分析程序路径、数据流、控制流等内部结构的基础上，设计测试用例和测试数据进行软件测试。

可测试性的度量方法依据白盒测试或黑盒测试策略，大体上分为 2 类。

(1) 以故障检测率为准则。如果一个软件在随机黑盒测试中能够暴露故障，并能对大多数故障输入产生失效，则认为该软件具有较高的可测试性；相反，在随机黑盒测试中难以检测到故障，对多数故障输入产生正确的输出，则认为该软件的可测试性较低。这种方法需要事先已知故障数。

(2) 以测试覆盖率为准则。如果一个软件能够以较少的测试路径数目达到测试覆盖要求，则认为该软件具有较高的可测试性；相反地，则认为该软件的可测试性较低。这里假设指定的测试覆盖率是可以达到的。

应该认识到，上述两种方法得到的结果可能是不一致的。举个例子，没有任何条件或循环表达式的直线式程序，如果以测试覆盖率为准则，它与具有复杂控制流的程序相比，具有更高的可测试性；但是，如果以故障检测率为准则，直线式程序与具有复杂控制流的程序相比，其可测试性很可能会更低。

## 1.5 软件测试模型

软件开发活动是一个过程，遵循特定的生存周期模型。无论什么样的软件开发生存周期模型，软件测试都是其中必不可少的环节，是贯穿整个生存周期的重要活动。本节通过对 V 模型、W 模型、H 模型等常见模型的分析，介绍软件测试在不同软件生存周期中的位置和起到的作用。

### 1.5.1 V 模型

V 模型最早于 20 世纪 80 年代后期由 Paul Rook 在软件开发瀑布模型的基础上提出，是最经典和具有代表意义的测试模型。V 模型反映了软件测试活动与软件需求分析和设计的关系，明确指出软件测试不仅仅是软件开发中的一个独立阶段，而应贯穿于整个软件开发生存周期之中。

V 模型分为左右两部分，如图 1-2 所示，左半部分描述基本的软件开发过程，按照箭头的方向从上到下分为不同的开发阶段；右半部分描述了与开发相对应的测试过程，按照箭头的方向自下而上分为不同的测试类别。右半部分中的软件测试类别与左半部分中的软件开

发阶段有着同等的重要性，在形状上呈现 V 字形，故称为 V 模型。V 模型中的左右两部分存在连通关系，使得不同的测试类别与相应的开发阶段对应，不仅说明了要做什么事，还说明了在什么时间和怎么样来实施这些任务。

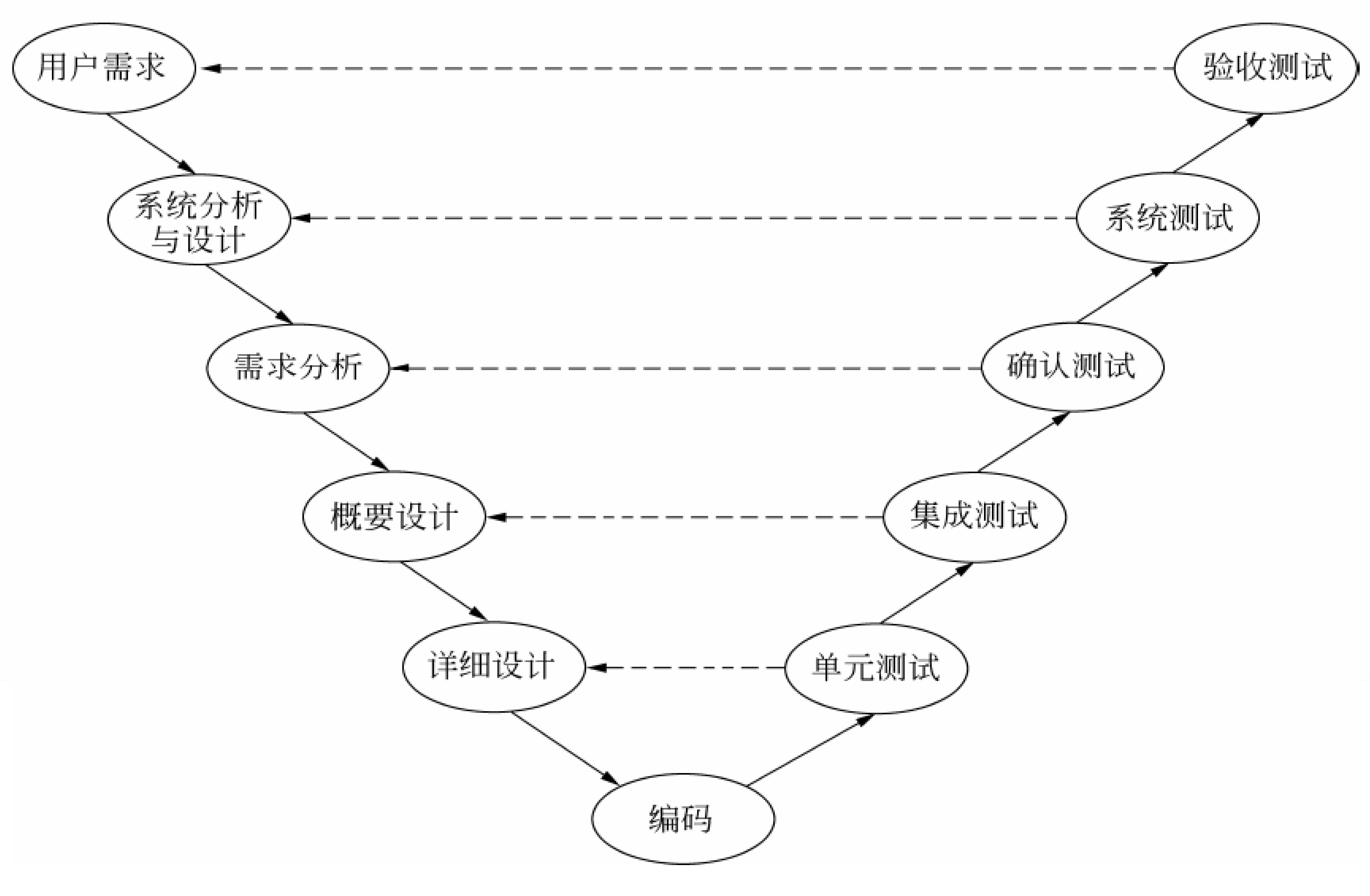



**图 1-2 V 模型示意图**

V 模型的主要不足在于：它把测试置于软件开发活动之后，对"尽早测试和不断测试"的原则体现不充分；它把不同类别的测试与软件开发阶段一一对应，但在实际项目中，各级测试与不同开发阶段很难有严格的对应关系；它容易使人理解为测试就是针对程序进行，缺少需求评审、设计评审、代码审查等验证确认与静态测试内容，有可能导致需求分析、概要设计等早期开发阶段的问题直至系统测试和验收测试才能被发现，错过最佳的缺陷发现和修复时机。

## 1.5.2 W 模型

W 模型是在 V 模型的基础上，由 Paul Herzlich 在 1999 年提出的。相对于 V 模型，W 模型增加了各软件开发阶段中同步进行的验证和确认活动。

W 模型由两个 V 字模型组成，如图 1-3 所示，开发过程是一个 V 字，伴随的测试过程是另一个 V 字，两者是并行关系。W 模型强调测试是伴随着整个软件开发周期的，而且测试对象不仅仅是程序，还应包括需求、设计等阶段的工作产品，也就是说，测试和开发是同步进行的。W 模型有助于尽早地、全面地发现问题。比如，在需求分析完成后，测试人员就应该参与到对需求的验证和确认活动中，以及时地找出存在的错误；同时，对需求的测试也有利于了解掌握项目情况和测试风险，尽早制定应对措施。

W 模型的不足主要在于：把软件开发和测试都看作是串行的活动，呈现出一种线性关系，等上一阶段完全结束后才可正式开始下一阶段的工作，无法支持迭代的开发模型，不能很好地适应软件开发复杂多变的情况。

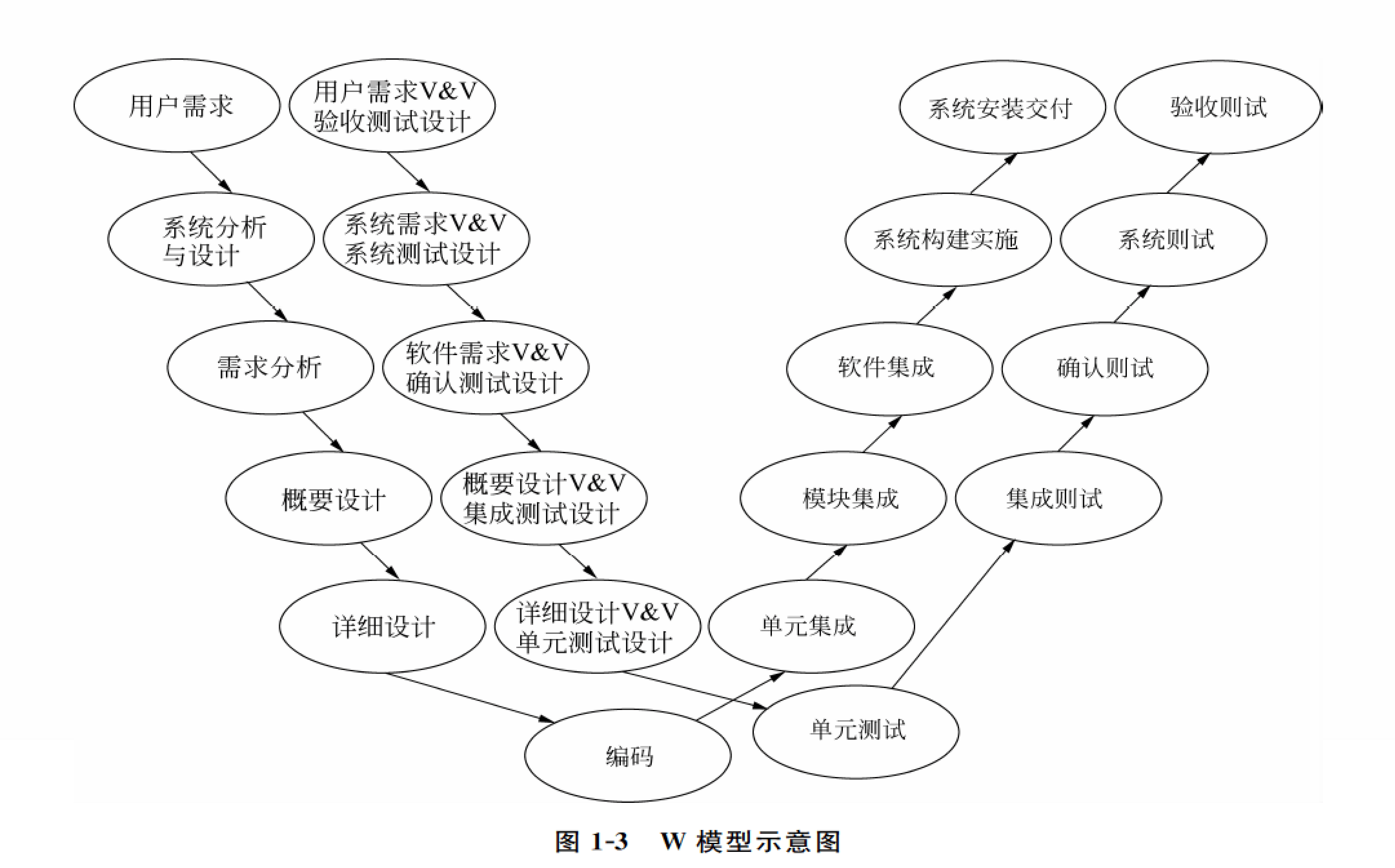



图 1-3 W 模型示意图

### 1.5.3 H 模型

V 模型和 W 模型把开发和测试看作存在严格次序的串行活动，与实际软件研制活动并不相符。为了克服 V 模型和 W 模型的上述不足，人们提出 H 模型，把测试活动独立出来，分为测试准备（包括测试分析、测试策划、测试设计等）和测试执行（包括测试执行、测试总结评估等）两个阶段活动，并形成一个与其他流程（比如软件开发中的设计流程）彼此独立的流程，如图 1-4 所示。

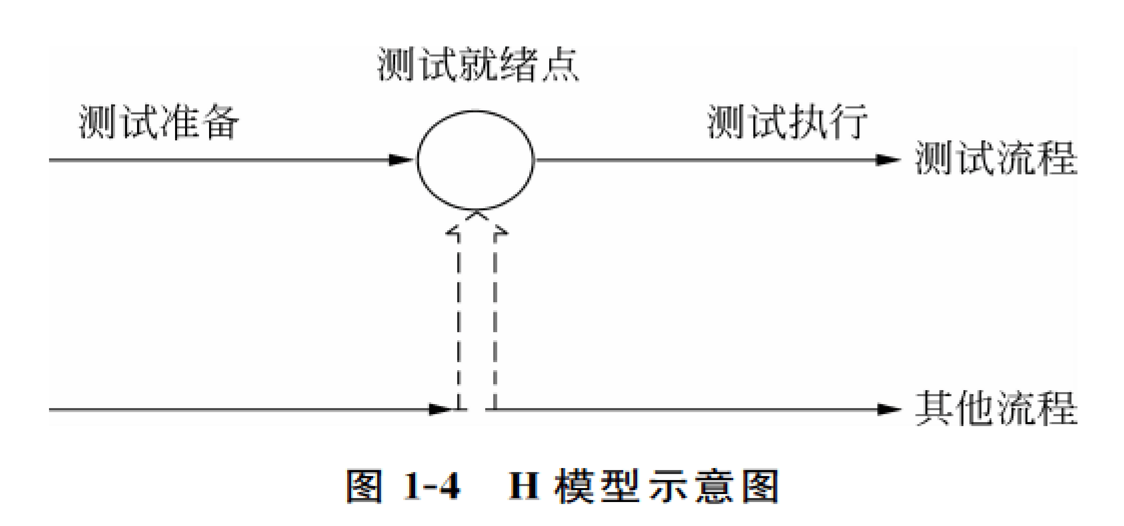



图 1-4 H 模型示意图

演示了在整个开发周期中某个时机对应的一次测试流程，标注的其他流程可以是任意的开发流程，比如分析流程、设计流程或编码流程等。整个软件测试可以包含很多这样的测试流程，并且不同的测试活动可以是按照某个次序先后进行的，也可以是反复的。只要测试条件成熟了，测试准备就绪了，就可以开展测试执行活动。

H 模型体现了测试流程的独立性、完整性和过程管理的重要性，也体现了“尽早测试和不断测试”的原则。

# 1.6 软件测试级别与测试类型

## 1.6.1 软件测试级别

在最常见的 V 模型和 W 模型中，把软件测试描述为单元测试、集成测试、确认测试、系统测试、验收测试等 5 种不同类别，这也是人们对软件测试级别最常用的划分方式。

按照 GB/T 15532—2008《计算机软件测试规范》，软件测试分为单元测试、集成测试、配置项测试（也称软件合格性测试或确认测试）、系统测试、验收测试等 5 种测试类别，可根据软件的规模、类型、完整性级别选择测试类别。

**1. 单元测试**

按照 GB/T 11457—2006《信息技术　软件工程术语》，单元测试是“独立的硬件或软件单元或相关单元的测试”。

GB/T 15532—2008 指出，单元测试的测试对象是可独立编译或汇编的程序模块（或称为软件构件或在面向对象设计中的类），测试目的是检查每个软件单元能否正确地实现设计说明中的功能、性能、接口和其他设计约束等要求，发现单元内可能存在的各种差错，一般由软件的供方或开发方组织并实施。

**2. 集成测试**

按照 GB/T 11457—2006《信息技术　软件工程术语》，集成测试是“把软件、硬件部件或两者结合起来进行的测试，并测试以评价它们之间的交互”。

GB/T 15532—2008 指出，软件集成测试的对象包括两种：任意一个软件单元集成到计算机软件系统的组装过程，以及任意一个组装得到的软件系统。集成测试的目的是检验软件单元之间、软件单元和已集成的软件系统之间的接口关系，并验证已集成软件系统是否符合设计要求，一般由软件供方组织并实施，测试人员与开发人员应相对独立。

**3. 配置项测试（也称软件合格性测试或确认测试）**

按照 GB/T 11457—2006《信息技术　软件工程术语》，合格性测试是“确定系统或部件是否适合于操作使用的测试行为”。

GB/T 15532—2008 指出，配置项测试的对象是软件配置项，软件配置项是为独立的配置管理而设计的并且能满足最终用户功能的一组软件，配置项测试的目的是检验软件配置项与软件需求规格说明的一致性。软件配置项测试一般由软件的供方组织，由独立于软件开发的人员实施，软件开发人员配合。如果配置项测试委托于第三方实施，一般应委托国家认可的第三方测试机构。

**4. 系统测试**

按照 GB/T 11457—2006《信息技术　软件工程术语》，系统测试是“在完整的、集成的系统上的测试行为，它用于评价系统与规定的需求的遵从性”。

GB/T 15532—2008 指出，系统测试对象是完整的、集成的计算机系统，重点是新开发的软件配置项的集合，系统测试的目的是在真实系统工作环境下检验完整的软件配置项能

否和系统正确连接，并满足系统/子系统设计文档和软件开发合同规定的要求。

**5. 验收测试**

按照 GB/T 11457—2006《信息技术 软件工程术语》，验收测试是“a)确定一系统是否符合其验收准则，使客户能确定是否接收此系统的正式测试；b)使用户、客户或其他授权实体确定是否接受系统或部件的正式测试”。

GB/T 15532—2008 指出，验收测试是以需方为主的测试，其对象是完整的、集成的计算机系统。验收测试的目的是在真实的用户(或称系统)工作环境下检验完整的软件系统，是否满足软件开发合同(或软件需求规格说明)规定的要求，其结论是软件需方确定是否接收该软件的主要依据。验收测试应由软件的需方组织，由独立于软件开发的人员实施。如果委托第三方实施，一般应委托国家认可的第三方测试机构。

单元测试、集成测试、配置项测试、系统测试、验收测试等不同测试级别或类别的测试策略、测试内容、测试方法、测试用例设计及测试过程等具体内容，将在本书后文中详细介绍。

## 1.6.2 软件测试类型

如 1.6.1 节所述，按照测试级别的不同，软件测试可分为单元测试、集成测试、确认测试、系统测试和验收测试。软件测试还有多种其他的分类方法。比如，从是否执行程序的角度，软件测试可以分为静态测试和动态测试；从是否关心软件内部结构的角度，软件测试可分为白盒测试和黑盒测试；从测试实施主体角度，软件测试可以分为开发方测试、用户方测试和第三方测试；从测试过程的自动化水平角度，软件测试可以分为人工测试、半自动化测试和自动化测试，等等。

从软件特性和质量目标的角度，软件测试可分为功能测试、性能测试、接口测试、人机交互界面测试、安全性测试等多种不同的测试类型。

**1. 功能测试**

功能测试是对软件需求规格说明或设计文档中的功能需求逐项进行的测试，以验证其功能是否满足要求。主要包括：

(1) 用正常值的等价类输入数据值测试；

(2) 用非正常值的等价类输入数据值测试；

(3) 进行每个功能的合法边界值和非法边界值输入的测试；

(4) 用一系列真实的数据类型和数据值运行，测试超负荷、饱和及其他“最坏情况”的结果；

(5) 在配置项测试时对配置项控制流程的正确性、合理性等进行验证。

**2. 性能测试**

性能测试是对软件需求规格说明或设计文档中规定的性能需求逐项进行的测试，以验证其性能是否满足要求。主要包括：

(1) 测试在获得定量结果时程序计算的精确性(处理精度)；

(2) 测试其时间特性和实际完成功能的时间(响应时间)；

(3) 测试为完成功能所处理的数据量；

(4) 测试程序运行所占用的空间；

(5) 测试其负荷潜力;
(6) 测试配置项各部分的协调性;
(7) 在系统测试时测试软件性能和硬件性能的集成;
(8) 在系统测试时测试系统对并发事件和并发用户访问的处理能力。

**3. 接口测试**

接口测试是对软件需求规格说明或设计文档的接口需求逐项进行的测试。主要包括:
(1) 测试所有外部接口,检查接口信息的格式及内容;
(2) 对每一个外部输入/输出接口必须做正常和异常情况的测试;
(3) 测试硬件提供的接口是否便于使用;
(4) 测试系统特性(如数据特性、错误特性、速度特性等)对软件功能、性能特性的影响;
(5) 对所有内部接口的功能、性能进行测试。

**4. 人机交互界面测试**

人机交互界面测试是对所有人机交互界面提供的操作和显示界面进行的测试,以检验是否满足用户的要求。主要包括:
(1) 测试操作和显示界面及界面风格与软件需求规格说明中要求的一致性和符合性;
(2) 以非常规操作、误操作、快速操作来检验人机界面的健壮性;
(3) 测试对错误命令或非法数据输入的检测能力与提示情况;
(4) 测试对错误操作流程的检测与提示;
(5) 对照用户手册或操作手册逐条进行操作和观察。

**5. 强度测试**

强度测试是强制软件运行在不正常到发生故障的情况下(设计的极限状态到超出极限状态),检验软件可以运行到何种程度的测试。主要包括:
(1) 提供最大处理的信息量;
(2) 提供数据能力的饱和实验指标;
(3) 提供最大存储范围(如常驻内存、缓冲、表格区、临时信息区);
(4) 在能力降级时进行测试;
(5) 在人为错误(如寄存器数据跳变、错误的接口)状态下进行软件反应的测试;
(6) 通过启动软件过载安全装置(如临界点警报、过载溢出功能、停止输入、取消低速设备等)生成必要条件,进行计算过载的饱和测试;
(7) 需进行持续一段规定的时间,而且连续不能中断的测试。

**6. 余量测试**

余量测试是对软件是否达到需求规格说明中要求的余量的测试。若无明确要求时,一般至少留有20%的余量。主要包括:
(1) 全部存储量的余量;
(2) 输入/输出及通道的吞吐能力余量;
(3) 功能处理时间的余量。

**7. 安全性测试**

安全性测试是检验软件中已存在的安全性、安全保密性措施是否有效的测试。测试应

尽可能在符合实际使用的条件下进行。主要包括：

(1) 对安全性关键的软件部件，必须单独测试安全性需求；

(2) 在测试中全面检验防止危险状态措施的有效性和每个危险状态下的反应；

(3) 对设计中用于提高安全性的结构、算法、容错、冗余及中断处理等方案，必须进行针对性测试；

(4) 对软件处于标准配置下其处理和保护能力的测试；

(5) 应进行对异常条件下系统/软件的处理和保护能力的测试(以表明不会因为可能的单个或多个输入错误而导致不安全状态)；

(6) 对输入故障模式的测试；

(7) 必须包含边界、界外及边界结合部的测试；

(8) 对“0”、穿越“0”以及从两个方向趋近于“0”的输入值的测试；

(9) 必须包括在最坏情况配置下的最小输入和最大输入数据率的测试；

(10) 对安全性关键的操作错误的测试；

(11) 对具有防止非法进入软件并保护软件的数据完整性能力的测试；

(12) 对双工切换、多机替换的正确性和连续性的测试；

(13) 对重要数据的抗非法访问能力的测试。

**8. 恢复性测试**

恢复性测试是对有恢复或重置功能的软件的每一类导致恢复或重置的情况逐一进行的测试，以验证其恢复或重置功能。恢复性测试是要证实在克服硬件故障后，系统能否正常地继续进行工作，且不对系统造成任何损害。主要包括：

(1) 探测错误功能的测试；

(2) 能否切换或自动启动备用硬件的测试；

(3) 在故障发生时能否保护正在运行的作业和系统状态的测试；

(4) 在系统恢复后，能否从最后记录下来的无错误状态开始继续执行作业的测试。

**9. 边界测试**

边界测试是对软件处在边界或端点情况下运行状态的测试。主要包括：

(1) 软件的输入域和输出域的边界或端点的测试；

(2) 状态转换的边界或端点的测试；

(3) 功能界限的边界或端点的测试；

(4) 性能界限的边界或端点的测试；

(5) 容量界限的边界或端点的测试。

**10. 互操作性测试**

互操作性测试是为验证不同软件之间的互操作能力而进行的测试。主要包括：

(1) 同时运行两个或多个不同的软件；

(2) 软件之间发生了互操作。

**11. 容量测试**

容量测试是检验软件的能力最高能达到什么程度的测试。容量测试一般应测试到在正常情况下软件所具备的最高能力，如响应时间或并发处理个数等能力。

**12. 安装性测试**

安装性测试是对安装过程是否符合安装规程的测试，以发现安装过程中的错误。主要包括：

（1）不同配置下的安装和卸载测试；

（2）安装规程的正确性测试。

### 1.6.3 软件关键等级

在大型软件系统中，往往包含很多软件，划分为不同的软件配置项。根据GB/T 11457—2006《信息技术 软件工程术语》，所谓的软件配置项是指“为配置管理设计的软件的集合，它在配置管理过程中，作为单个实体对待”。

为了更加有效地实施管理以及工程上的需要，人们经常对软件配置项进行分级管理，对不同等级的软件配置项实施差异化管理。影响软件配置项关键等级的因素包括软件配置项在软件系统中地位、起到的作用以及其他相关因素等。在航天测量控制系统中，根据软件配置项失效所造成的危险的严重程度、危险发生的概率、软件对危险的控制程度以及控制的复杂性和实时性等，把软件配置项分为A、B、C、D四个关键等级。这里所说的危险，既包括影响系统和人员生命的危险，也包括影响任务成败的危险。

软件关键等级的确定过程按如下5步进行。

（1）确定各系统和各分系统的所有潜在危险。软件系统分析与设计阶段应开展系统的危险分析，系统危险分析是系统中各层次开展危险分析的前提和基础。通过系统的危险分析，列出系统所有的潜在危险，包括危险的原因以及针对危险原因的危险控制等。系统危险分析是确定软件关键等级的前提和基础。

（2）分析并确定每一种潜在危险的严重等级和发生的可能性。危险的严重等级和发生的可能性主要是基于工程的判断得出。危险的严重等级定义为灾难的、严重的、一般的、可忽略的等4级，详见表1-1；危险发生的可能性定义为非常可能、很可能、可能、不太可能、几乎不可能等5级，详见表1-2。

**表1-1 危险的严重等级**

| 危险严重等级 | 定义描述 |
|---|---|
| 灾难的 | 人员死亡；系统报废；基本任务失败 |
| 严重的 | 人员严重受伤；系统严重损坏；基本任务的主要部分未完成 |
| 一般的 | 人员轻度受伤；系统轻度损坏；对完成任务有轻度影响 |
| 可忽略的 | 轻于轻度的人员伤害或系统损坏；执行任务中有障碍，但不影响完成任务 |

**表1-2 危险发生的可能性级别**

| 发生的可能性定义 | 定义描述 |
|---|---|
| 非常可能 | 危险将频繁发生。$P>10^{-1}$ |
| 很可能 | 危险将发生数次。$10^{-2}< P\leqslant 10^{-1}$ |

续表

| 发生的可能性定义 | 定义描述 |
|---|---|
| 可能 | 某时候危险很可能发生。$10^{-3} < P \leqslant 10^{-2}$ |
| 不太可能 | 危险发生的可能性很小。$10^{-4} < P \leqslant 10^{-3}$ |
| 几乎不可能 | 危险几乎不可能发生。$P \leqslant 10^{-4}$ |

说明：$P$ 是危险发生的概率。危险发生的可能性可以定性的表示，也可以用定量的概率来表示。但它一般是基于工程判断得出的，而不是基于确实的数字。

(3) 确定系统风险指标。根据每种危险的严重等级及其发生的可能性，确定每种危险的系统风险指标，如表 1-3 所示。在确定资源分配和考虑能否接受某种风险指标时，系统风险指标是一个重要依据。最高风险的危险(系统风险指标为"1")在系统分析与设计中是不允许存在的。对于系统风险指标为"1"的系统分析与设计应进行重新设计以消除危险产生的概率或使之降低到可接受的范围；系统风险指标为"6"和"7"可以不进行安全性分析或控制；系统风险指标为"5"要求最少的安全分析或控制；对于系统风险指标为"2"、"3"和"4"，所要求的安全分析工作随着系统风险指标的增高而增加。

**表 1-3 危险的系统风险指标**

| 危险可能性级别 / 危险严重等级 | 非常可能 | 很可能 | 可能 | 不太可能 | 几乎不可能 |
|---|---|---|---|---|---|
| 灾难的 | 1 | 1 | 2 | 3 | 4 |
| 严重的 | 1 | 2 | 3 | 4 | 5 |
| 一般的 | 2 | 3 | 4 | 5 | 6 |
| 可忽略的 | 3 | 4 | 5 | 6 | 7 |

(4) 确定软件的控制类别。通过系统危险分析，列出系统所有的潜在危险，包括危险的原因以及针对危险原因的危险控制等。其中有一些软件就是直接导致危险的原因，而有些软件则是用来控制、缓解或检测危险的。软件系统危险分析的目的就是要将这些与软件相关的危险以及与这些危险相关的软件标识出来。对于这些标识出来的软件要逐一确定其控制类别。软件的控制类别与软件对系统的控制程度、控制的复杂性和实时性等有关。软件的控制类别定义为 4 类，详细定义见表 1-4。

**表 1-4 软件的控制类别**

| 软件控制类别 | 描述 | |
|---|---|---|
| Ⅰ | 1 | 软件对安全关键功能进行部分或全部自主控制 |
| | 2 | 包含多个子系统、交互作用的并行处理器或多个接口的复杂系统 |
| | 3 | 有一些或全部安全关键功能是时间关键的 |
| Ⅱ | 1 | 控制危险，但是其他安全系统可以进行部分缓解<br>检测危险，需要采取安全措施时通知操作员 |
| | 2 | 包含少量子系统和(或)一些接口的中等复杂系统，无并行处理 |
| | 3 | 有一些危险控制动作可能是时间关键的，具有临界时间要求，但是不会超过合适的操作员所需要的时间或自动系统响应的时间 |

续表

| 软件控制类别 | | 描　述 |
|---|---|---|
| Ⅲ | 1 | 若软件发生故障,则有一个或若干个缓解系统防止危险发生<br>提供冗余的安全关键信息资源 |
| | 2 | 稍微复杂的系统,仅包含有限数目的接口 |
| | 3 | 缓解系统能在任何临界时间内进行响应 |
| Ⅳ | 1 | 对危险无控制<br>不为操作员提供安全关键数据 |
| | 2 | 仅带 2～3 个子系统的简单系统,仅包含少量接口 |
| | 3 | 没有临界时间要求 |

注:每个软件控制类别有 3 种描述,它们之间是“或”的关系,即只要符合 1 种,就可以确定。

(5) 确定每个软件的关键等级。根据每个软件的控制类别和系统风险指标确定每个软件的关键等级。软件关键等级分为 A、B、C、D 四个等级,详细定义见表 1-5。

**表 1-5　软件的关键等级**

| 软件控制类别 \ 关键等级 \ 系统风险指标 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Ⅰ | A | B | C | D |
| Ⅱ | B | C* | D | D |
| Ⅲ | C* | D | D | D |
| Ⅳ | D | D | D | D |

* 对该软件关键等级的确定应仔细分析。若某个软件有多个等级,则按其中最高的等级确定。

# 1.7　软件测试标准规范

## 1.7.1　相关标准概述

与在工程技术领域一样,标准在信息技术和软件工程领域也发挥着巨大的作用。在国家标准化管理委员会、ISO 以及 IEEE 的网站上,能够查询到大量的软件测试相关标准,其中有直接涉及软件测试的,更多的是信息技术、软件工程或软件质量等方面与软件测试相关的标准。主要包括以下 25 种。

(1) GB/T 8566—2007《信息技术　软件生存周期过程》。本标准修改采用国际标准 ISO/IEC 12207:1995《信息技术　软件生成周期过程》和 ISO/IEC 12207:1995/Amd. 1:2002 以及 ISO/IEC 12207:1995/Amd. 2:2004(英文版)。本标准为软件生存周期过程建立了一个公共框架,以供软件产业界使用。它包含在含有软件的系统、独立软件产品和软件服务的获取期间以及在软件产品的供应、开发、运行和维护期间需应用的过程、活动和任务。

(2) GB/T 8567—2006《计算机软件文档编制规范》。本标准是 GB/T 8567—1988《计算机软件产品开发文件编制指南》的修订版。主要对软件的开发过程和管理过程应编制的主要文档及其编制的内容、格式规定了基本要求。

(3) GB/T 9385—2008《计算机软件需求说明编制指南》。本标准给出了软件需求规格说明的编制要求，描述了一份好的软件需求规格说明的内容和质量，并在附录中给出了一些软件需求规格说明的提纲示例。

(4) GB/T 9386—2008《计算机软件测试文件编制规范》。本标准规定了一组基本的计算机软件测试文档的格式和内容要求，适用于计算机软件生存周期全过程。

(5) GB/T 11457—2006《软件工程术语》。本标准定义软件工程领域中通用的术语，适用于软件开发、使用维护、科研、教学和出版等方面。

(6) GB/T 14394—2008《计算机软件可靠性和可维护性管理》。本标准规定了软件产品在其生存周期内如何选择适当的软件可靠性和可维护性管理要求，并指导软件可靠性大纲和可维护性大纲的制定和实施。

(7) GB/T 15532—2008《计算机软件测试规范》。本标准替代了 GB/T 15532—1995《计算机软件单元测试》。本标准规定了计算机软件生存周期内各类软件产品的基本测试方法、过程和准则。

(8) GB/T 16260.1—2006《软件工程　产品质量　第 1 部分：质量模型》。本标准等同采用 ISO/IEC 9126-1：2001《软件工程　产品质量　第 1 部分：质量模型》，描述了关于软件产品质量的两部分模型：内部质量和外部质量、使用质量。

(9) GB/T 16260.2—2006《软件工程　产品质量　第 2 部分：外部度量》。本标准等同采用 ISO/IEC 9126-2：2003《软件工程　产品质量　第 2 部分：外部度量》，定义了依据 GB/T 16260.1—2006 标准定义的特性和子特性来定量测量软件外部质量的外部度量。

(10) GB/T 16260.3—2006《软件工程　产品质量　第 3 部分：内部度量》。本标准等同采用 ISO/IEC 9126-3：2003《软件工程　产品质量　第 3 部分：内部度量》，定义了依据 GB/T 16260.1—2006 标准定义的特性和子特性来定量测量软件内部质量的内部度量。

(11) GB/T 16260.4—2006《软件工程　产品质量　第 4 部分：使用质量的度量》。本标准等同采用 ISO/IEC 9126-4：2004《软件工程　产品质量　第 4 部分：使用质量的度量》，为 GB/T 16260.1—2006 中规定的质量特性定义了使用质量的度量。

(12) GB/T 16680—1996《软件文档管理指南》。本标准非等效采用国际标准 ISO/IEC TR 9294：1990《信息技术　软件文档管理指南》，为那些对软件或基于软件的产品的开发负有职责的管理者提供软件文档的管理指南，目的是协助管理者在他们的机构中产生有效的文档。

(13) GB 17859—1999《计算机信息系统安全保护等级划分准则》。本标准适用于计算机信息系统安全保护技术能力等级的划分。本标准规定了计算机信息系统安全保护能力的五个等级，由低至高分别为：用户自主保护级、系统审计保护级、安全标记保护级、结构化保护级、访问验证保护级。

(14) GB/T 18221—2000《信息技术　程序设计语言、环境与系统软件接口　独立于语言的数据类型》。本标准等同采用国际标准 ISO/IEC 11404：1996《信息技术　程序设计语言、环境与系统软件接口　独立于语言的数据类型》。本标准规定了程序设计语言和软件接口中通用的一批数据类型的术语和共享语义，称之为独立于语言(LI)的数据类型。

(15) GB/T 18234—2000《信息技术　CASE 工具的评价与选择指南》。本标准等同采用国际标准 ISO/IEC 14102：1995《信息技术　CASE 工具的评价与选择指南》，定义了用于对某个 CASE 工具进行技术评价和最终选择的一系列过程及一组结构化的 CASE 工具特性。

(16) GB/T 18491.1—2001《信息技术　软件测量　功能规模测量　第 1 部分：概念定义》。本标准等同采用国际标准 ISO/IEC 14143.1：1998《信息技术　软件测量　功能规模测量　第 1 部分：概念定义》，定义了功能规模测量(FSM)的主要概念，描述了应用 FSM 方法的一般原则。

(17) GB/T 18492—2001《信息技术　系统及软件完整性级别》。本标准等同采用国际标准 ISO/IEC 15026：1998《信息技术　系统及软件完整性级别》，定义了与完整性级别相关的概念，定义了确定完整性级别和软件完整性需求的过程，并提出对每个过程的需求。

(18) GB/T 18905.1—2002《软件工程　产品评价　第 1 部分：概述》。本标准等同采用国际标准 ISO/IEC 14598-1：1999《信息技术　软件产品评价　第 1 部分：概述》(英文版)。本标准介绍了 GB/T 18905 的其他部分的概述，给出了技术术语，解释了 GB/T 18905 与 ISO/IEC 9126 标准中质量模型的关系。

(19) GB/T 18905.2—2002《软件工程　产品评价　第 2 部分：策划和管理》。本标准等同采用国际标准 ISO/IEC 14598-2：2000《软件工程　产品评价　第 2 部分：策划和管理》(英文版)，详细地提供了有关软件产品评价的策划和管理需求。

(20) GB/T 18905.3—2002《软件工程　产品评价　第 3 部分：开发者用的过程》。本标准等同采用国际标准 ISO/IEC 14598-3：2000《软件工程　产品评价　第 3 部分：开发者用的过程》(英文版)，适用于所有需要严格要求过程的软件开发活动，供软件开发期间使用。

(21) GB/T 18905.4—2002《软件工程　产品评价　第 4 部分：需方用的过程》。本标准等同采用国际标准 ISO/IEC 14598-4：1999《软件工程　产品评价　第 4 部分：需方用的过程》(英文版)，包含了在获取现货软件产品、定制的软件产品或修改现有的软件产品时，对软件产品质量进行系统地测量、评估和评价的需求、建议和指南。

(22) GB/T 18905.5—2002《软件工程　产品评价　第 5 部分：评价者用的过程》。本标准等同采用国际标准 ISO/IEC 14598-5：1999《软件工程　产品评价　第 5 部分：评价者用的过程》(英文版)，定义了分析各类软件产品的评价需求，规定、设计和实施评价，并对评价做出结论所需的各种活动。

(23) GB/T 18905.6—2002《软件工程　产品评价　第 6 部分：评价模块的文档编制》。本标准等同采用国际标准 ISO/IEC 14598-6：2001《软件工程　产品评价　第 6 部分：评价模块的文档》(英文版)，定义了用于描述评价模块的文档编制的结构和内容。

(24) GB/Z 18914—2002《信息技术　软件工程 CASE 工具的采用指南》。本标准等同采用国际标准 ISO/IEC TR 14471：1999(英文版)，阐述了针对 CASE 工具的产品评价、选择和采用方面的问题，是对涉及这些问题一般方面的有关国家标准的补充。

(25) GB/T 19000—2008《质量管理体系　基础和术语》。本标准等同采用 ISO 9000：2005《质量管理体系　基础和术语》，是 GB/T 19000 族的核心标准之一，表述了构成 GB/T 19000 族标准主体内容的质量管理体系的基础，并定义了相关的术语。

### 1.7.2　GB/T 9386—2008《计算机软件测试文档编制规范》

GB/T 9386—2008《计算机软件测试文档编制规范》描述了一组与软件测试实施方面有关的基本测试文档，规定了各个测试文档的目的、格式和内容。这些测试文档包括测试计划、测试说明（包括测试设计说明、测试用例说明和测试规程说明）和测试报告（包括测试项传递报告、测试日志、测试事件报告和测试总结报告）等。

GB/T 9386—2008 是在 GB/T 9386—1988 的基础上进行的第一次修订。与 1988 版相比，2008 版标准的名称和核心内容都没有改变，主要是增加了 10 条测试术语的定义，调整了部分章节编排方式，扩充了部分内容，并增加了 2 个作为资料性附录的文档编写示例。

GB/T 9386—1988 是基于 IEEE Std 829—1983 *Standard for Software Test Documentation* 编制的，而 IEEE Std 829—1983 已被 IEEE Std 829—2008 *Standard for Software and System Test Documentation* 所替代。目前 GB/T 9386 中尚未体现 IEEE Std 829—2008 对 IEEE Std 829—1983 的修订内容。

### 1.7.3　GB/T 15532—2008《计算机软件测试规范》

GB/T 15532—2008《计算机软件测试规范》规定了计算机软件生存周期内各类软件产品的基本测试方法、过程和准则。GB/T 15532—2008 对单元测试、集成测试、配置项测试（也称软件合格性测试或确认测试）、系统测试和验收测试等 5 种测试类别作了详细描述，每种测试类别按照测试对象和目的、测试的组织和管理、技术要求、测试内容、测试环境、测试方法、准入条件、准出条件、测试过程和输出文档等条目进行说明。

GB/T 15532—2008《计算机软件测试规范》替代了 GB/T 15532—1995《计算机软件单元测试》。GB/T 15532—1995 是基于 IEEE std 1008—1987 *Standard for Software Unit Testing* 编制的，GB/T 15532—2008 对 GB/T 15532—1995 进行了全面扩充。两个版本所描述的范围差异很大，2008 版修改了标准的名称，对典型的软件测试过程提出了全面的规范要求，从章节编排到规范内容与 1995 版相比有很大的发展。

### 1.7.4　GB/T 25000.51—2010《软件工程软件产品质量要求和评价（SQuaRE）商业现货（COTS）软件产品的质量要求和测试细则》

GB/T 25000.51—2010《软件工程软件产品质量要求和评价（SQuaRE）商业现货（COTS）软件产品的质量要求和测试细则》为软件特别是商业现货（COTS）软件规定了基本的质量要求和测试方法，以便于软件的供应、选择、采购和测试。

COTS 软件产品是一种打包出售的现货产品，典型情况是，这种软件产品与其用户文档集一起预先包装好出售。需方对其特征和其他质量没有任何影响，包装封面上提供出的信息常常是制造商或营销组织能与需方或用户交流的唯一手段。因此，把实质性信息提供给需方使其能按自己需要来评价 COTS 软件产品的质量是必要的。

由于 COTS 软件产品可能要在各种环境中运行，并且用户是在没有机会与类似产品作性能比较的情况下就作出选择，因此选用高质量的 COTS 软件产品是极其重要的。供方需要一种方式以确保 COTS 软件产品给予用户服务信用。一些供方可能选择第三方的评价或认证，以协助其提供这种信用。当用户要求确保避免业务或安全攸关的风险时，这种保证可能需要由用户在采购后选用特定的技术来处置。本标准宗旨并不在对 COTS 规定最低限度的业务或安全攸关的质量要求，不过是给出了资料性指南。

GB/T 25000.51—2010 是 GB/T 17544—1998《信息技术软件包质量要求和测试》的修订版。其修订的目的是为了与 SQuaRE 系列标准保持一致。本标准与 GB/T 17544—1998 的差异主要是：对结构作了调整和修改，GB/T 17544—1998 的篇幅共 4 章 3 个附录，新版标准共 7 章 3 个附录和 1 个参考文献；对内容进行了修改，对适用范围作了扩充。

## 1.7.5 ISO/IEC 29119 *Software Testing*

目前，现行的直接涉及软件测试的国际标准或国外先进标准主要有：由电气和电子工程师协会 IEEE(Institute of Electrical and Electronics Engineers)发布的 IEEE Std 829—2008 *Standard for Software and System Test Documentation* 和 IEEE Std 1008—1997 (R2002) *Standard For Software Unit Testing*，以及由英国标准协会 BSI(British Standards Institution)发布的 BS 7925-1 *Vocabulary of Terms in Software Testing* 与 BS 7925-2 *Software Component Testing Standard*。

最新的国际标准 ISO/IEC 29119 *Software Testing* 已发布，该标准由国际标准化组织 ISO(International Organization for Standardization)及国际电工委员会 IEC(International Electrotechnical Commission)组织制定，由 ISO/IEC JTC1/SC7/WG26 负责。ISO/IEC 29119 *Software Testing* 由 5 部分组成，如图 1-5 所示，包括：

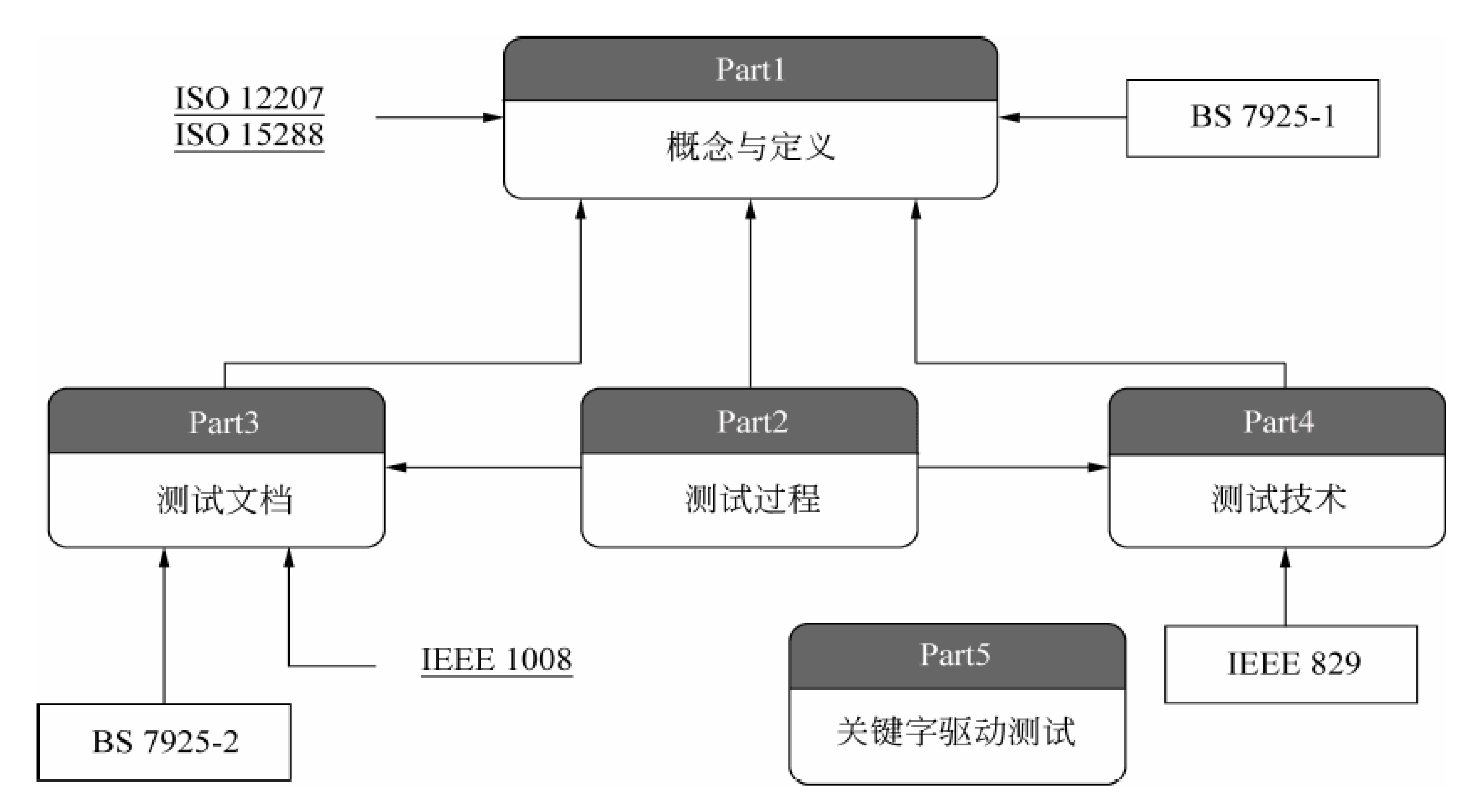



图 1-5 ISO/IEC 29119 的组成

(1) ISO/IEC 29119-1：*Concepts & Definitions*，2013 年 9 月发布；

(2) ISO/IEC 29119-2：*Test Processes*，2013 年 9 月发布；

（3）ISO/IEC 29119-3：*Test Documentation*，2013 年 9 月发布；

（4）ISO/IEC 29119-4：*Test Techniques*，2015 年 12 月发布；

（5）ISO/IEC 29119-5：*Keyword-Driven Testing*，2016 年 11 月发布。

**1. ISO/IEC 29119-1 概念与术语**

ISO/IEC 29119-1 介绍了 29119 系列标准中使用的术语，提供了概念应用的范例，目的是方便于对整个 29119 系列标准的理解和使用。本标准替代 BS 7925-1 *Vocabulary of Terms in Software Testing*。

ISO/IEC 29119-1 的主要章节内容包括：

（1）软件测试介绍；

（2）组织和项目中的软件测试；

（3）软件生存周期中的一般测试过程；

（4）基于风险的测试；

（5）测试子过程；

（6）测试实践；

（7）测试中的自动化；

（8）缺陷管理；

（9）测试在验证与确认中的角色、指标与度量、不同生存周期模型中的测试。

**2. ISO/IEC 29119-2 测试过程**

ISO/IEC 29119-2 定义了一个能用于任何软件开发生存周期的通用测试过程模型，该过程模型由组织级测试（例如：组织级测试方针、组织级测试策略等）、测试管理、动态测试等 3 个层次组成，如图 1-6、图 1-7 和图 1-8 所示。

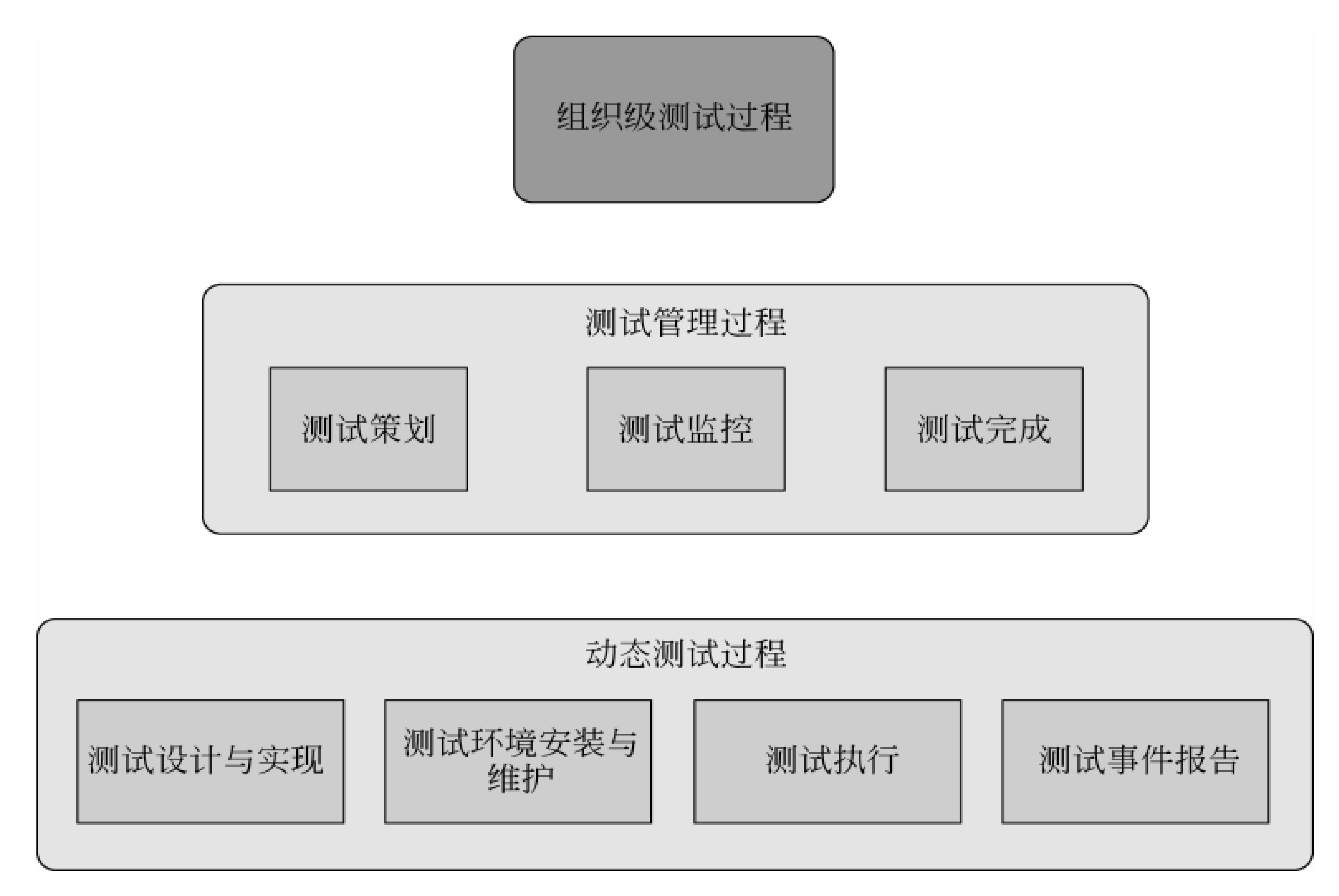



**图 1-6　测试过程模型**

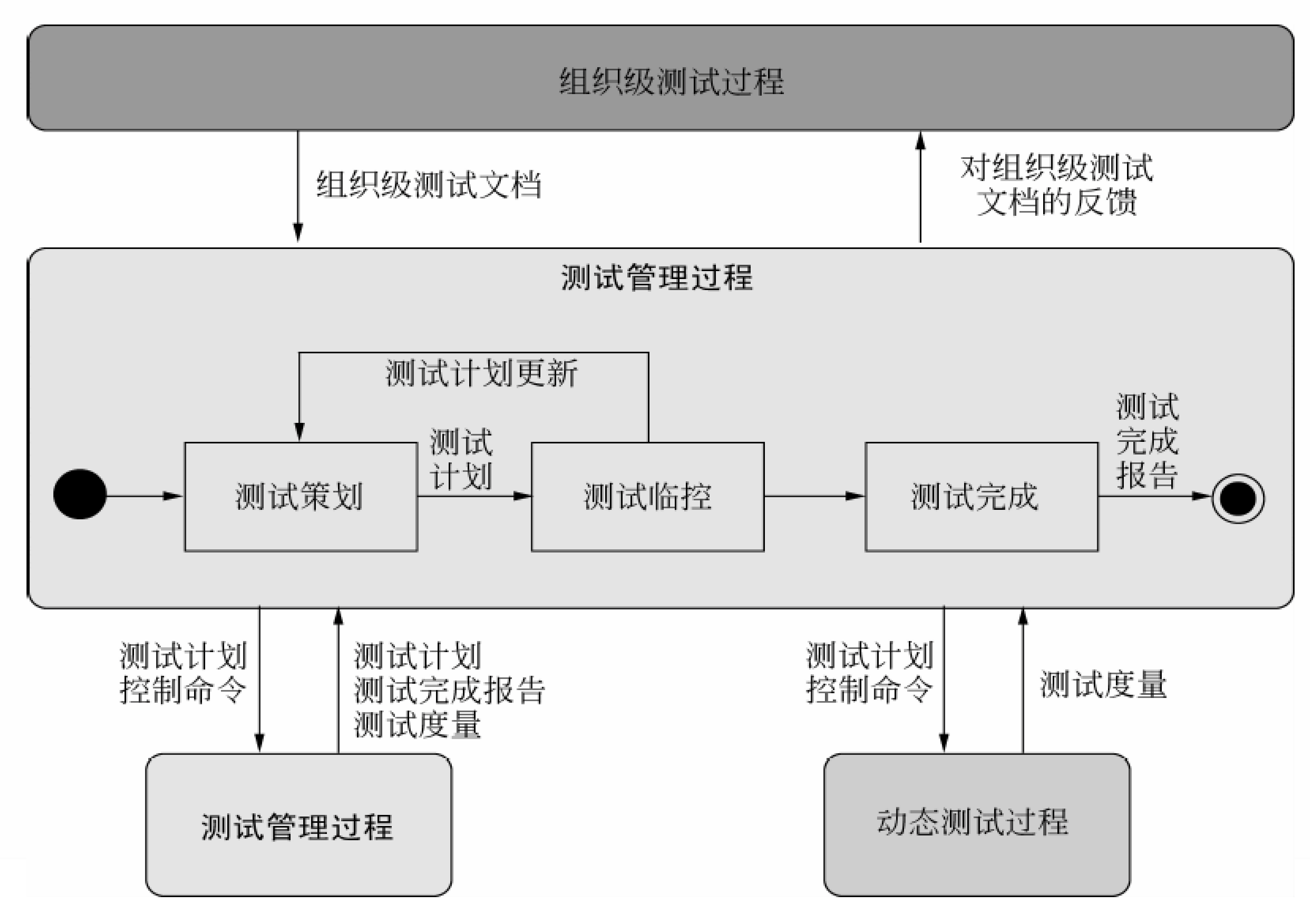



图 1-7 测试管理过程

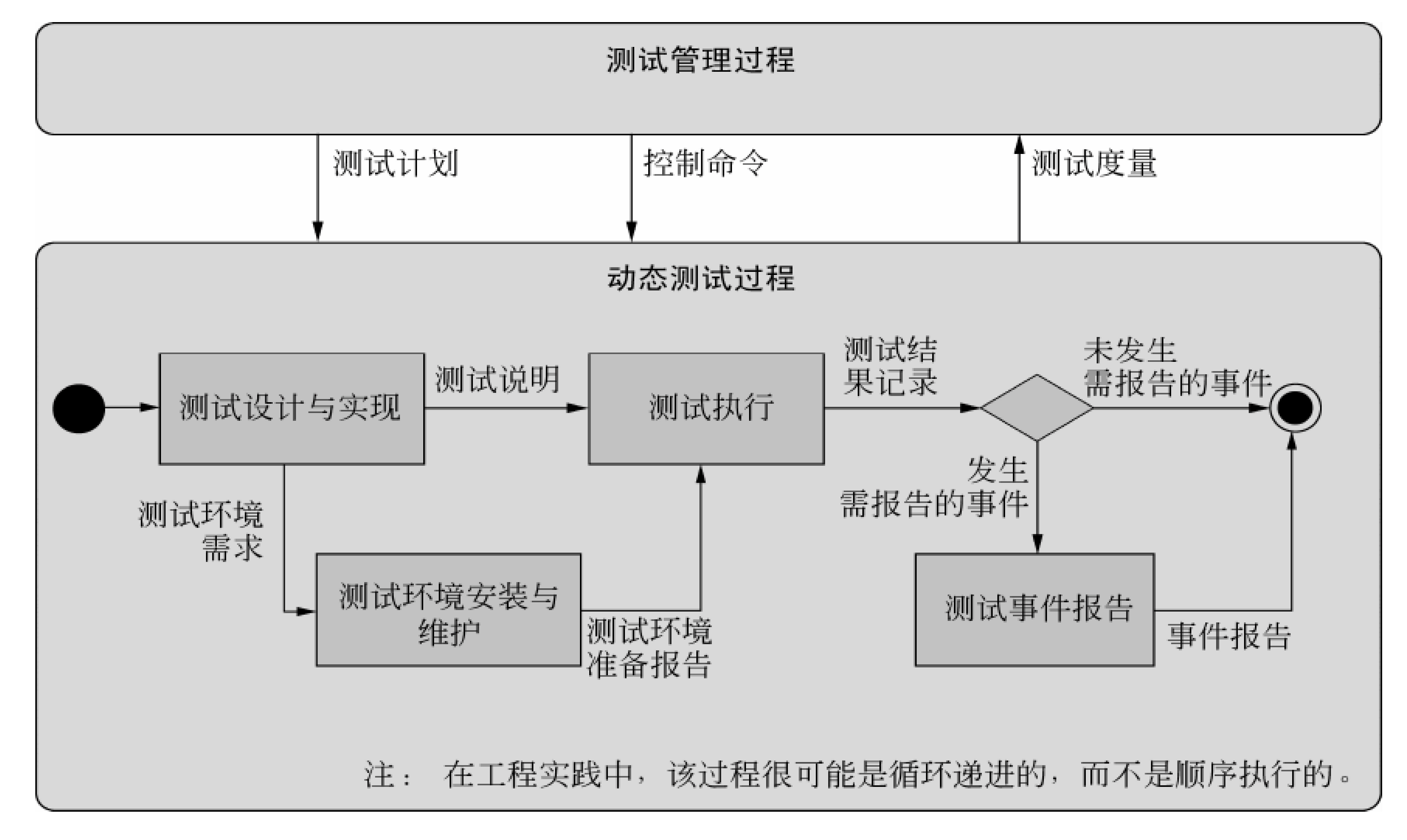



图 1-8 动态测试过程

**3. ISO/IEC 29119-3 测试文档**

ISO/IEC 29119-3 的目的是定义覆盖整个软件测试生存周期的测试文档模板。每个模板可定制，以满足不同组织在标准实施时的独特需求以及支持不同的软件开发生存周期模型。所有模板都与 ISO/IEC 29119-2 中定义的测试过程相匹配，并可在测试过程的应用中得到。本标准替代 IEEE 829。

ISO/IEC 29119-3 中定义的文档主要包括：

(1) 组织级测试过程文档,包括测试方针、组织级测试策略等;

(2) 测试管理过程文档,包括测试计划(含测试策略)、测试状态报告、测试完成情况报告等;

(3) 动态测试过程文档,包括测试设计说明、测试用例说明、测试规程说明、测试数据需求说明、测试数据就绪报告、测试环境需求说明、测试环境就绪报告、测试实测记录、测试总结报告、测试执行日志、测试问题报告等。

**4. ISO/IEC 29119-4 测试技术**

ISO/IEC 29119-4 的目的是定义一个覆盖软件测试技术(或称测试用例设计技术或测试方法)的国际标准,这些测试技术可用于任何组织或软件开发生存周期模型中的测试设计与实现阶段。该部分将以 BS 7925-2 为基础编制。

该标准包含的测试技术主要有:

(1) 基于需求的测试技术:等价类划分、分类树方法、边界值分析、状态转换测试、决策表测试、因果图分析、语法测试、组合测试、情景图测试、随机测试。

(2) 基于结构的测试技术:语句测试分支测试判断测试条件测试数据流测试。

(3) 基于经验的测试技术:错误猜测。

标准给出了各种测试技术的覆盖率度量指标。另外,该标准还给出一些测试类型的资料性定义,以及如何在这些测试类型中应用上述测试技术的示例。测试类型包括可用性测试、可靠性测试、安全性测试、稳定性测试、可移植性测试、性能测试(例如负载测试、压力测试、强度测试)、可维护性测试、本地化测试、安装性测试、互操作性测试、容灾测试、恢复性测试等。

**5. ISO/IEC 29119-5 关键字驱动测试**

ISO/IEC 29119-5 的目的是定义一个支持关键字驱动测试的国际标准。关键字驱动测试是一种利用事先定义好的关键字集合描述测试用例的方法。关键字描述了执行测试用例特定步骤的一系列操作的名称。通过使用关键字而不是自然语言描述测试步骤,更易于测试用例的理解、维护和自动化执行。

该标准已于 2016 年 11 月发布,主要包括如下内容:

(1) 关键字驱动测试简介,主要介绍关键字驱动测试的优势,关键字是如何被分层组织的,关键字的一般类型以及如何与数据建立关联等;

(2) 关键字驱动测试应用,讨论关键字怎样被识别,如何被用于组成测试用例,维护已定义的关键字集合应注意什么事项,以及关键字驱动测试和数据驱动测试为什么是相关的等;

(3) 角色与任务,描述关键字驱动测试中的角色及其任务,项目组成员可根据其与任务相匹配的资质担任一个或多个角色;

(4) 关键字驱动测试框架,关键字驱动测试可被由诸如商业工具、个人脚本和文档等所构成的框架来支撑,讨论一个适当的关键字驱动测试框架是如何被构建的,其哪些性质是必须的、哪些是有益的;

(5) 数据交互,介绍在关键字驱动测试工具之间需要交互哪些数据,给出一种数据格式描述。

# 1.8 软件测试人员能力素质要求

人们甚至部分软件测试从业者对软件测试经常存在的 4 种错误观点。

(1) 软件测试不可能发现所有的错误,没必要较真。软件测试确实不容易发现软件中存在的全部错误,但是软件测试发现的错误一般都是最容易发生的错误,并且即使只排除了几个错误,就可能会避免巨大的损失,带来可观的回报。

(2) 软件测试工作琐碎而枯燥,没有创造性。软件测试需要逐步逐个地梳理测试需求,对每个测试需求分解若干个测试项,对每个测试项设计若干个测试用例,然后要准备详细的测试数据,逐个地执行测试用例,记录测试实际结果,分析其与期望结果的一致性等,工作确实比较琐碎,但是无论是测试需求的梳理、测试用例的设计还是测试结果的分析,都需要测试人员具有创造性才能更好更高效地完成测试工作。

(3) 软件测试人员了解的软件背景知识没有设计人员或开发人员掌握的更多更深,缺少自信心。好的软件测试是测试技术、背景知识和测试经验的完美结合,好的测试人员能够综合各种技能多快好省地发现软件缺陷,最大限度地提高测试效果和效率,能够创造巨大的价值。

(4) 软件测试工作就是给人挑毛病的,容易招人讨厌。软件测试是站在委托方的角度,努力发现软件中存在的问题,最大程度地提升软件质量,保证软件在最终用户手中少出问题,事实上也是为开发方挽回或减少损失。另外,软件测试对事不对人,发现错误并不针对编程人员,而是尽量对整个软件产品有益,测试人员和编程开发人员的立场从本质上是一致的。

在对软件测试有正确认识的基础上,软件测试人员的能力素质模型主要包括如下 4 个方面。

(1) 良好的沟通能力。由于工作性质的需要,软件测试人员需要经常与用户和开发方、技术或非技术人员等不同类型人员打交道,必须具备良好的与人沟通的能力。测试工程师既要与用户谈得来,又要与开发人员说得上话。和用户交流时,重点须放在系统可以正确地处理什么和不可以处理什么,尽量不使用专业术语。和开发方交流时,要尽量使用专业术语,对相同的信息,软件测试人员须重新组织以另一种方式表达出来。

软件测试人员要善于表达自己的观点,一方面,要坚定地表明软件缺陷为何必须修复,并通过实际演示力陈观点;另一方面,要采用委婉的态度和适当的方式,使得开发方和用户愿意接受,特别是尽量避免与开发人员可能的冲突。

(2) 扎实的工作作风。软件测试人员在工作中要做到"五心"。一是专心:要集中精力,不可一心二用。精力集中不仅能够提高工作效率,还能发现更多的软件缺陷。二是细心:软件测试须认真细致执行,不忽略关键细节。如果不细心,有些软件缺陷将很难被发现。三是耐心:很多时候软件测试会显得非常枯燥,需要很大的耐心才能做好。如果比较浮躁,也不会做到专心和细心,很多缺陷将从眼前逃掉。四是责任心:责任心是做好任何工作的必备素质之一,软件测试尤其如此。软件测试往往起到最后把关的作用,如果敷衍了事,软件缺陷就会被放进发布版本或最终应用中,很可能造成非常严重的后果。五是自信

心：自信心是很多测试人员缺少的一项素质，遇到困难缩手缩脚，工作质量大打折扣。但只有具备了较强的自信心，才能更好地与用户和开发方沟通，才能更好地开展测试工作和发现软件缺陷。软件测试人员须建立能解决一切测试问题的信心。

（3）全面的技术基础。软件测试人员需要具备较为全面的技术，才能高质量地完成测试工作，才能更好地与开发方进行沟通和交流。除了设计测试用例、编写测试脚本、使用测试工具、建立测试环境等测试技术外，技术基础通常还包括编程语言、系统架构、操作系统、网络通信、数据库的特性与操作等，还需要了解被测软件系统的背景知识、清楚被测软件用到的概念和技术等。

（4）高级的综合素质。要成为一名优秀的软件测试人员，还应具备一些高级的能力素质，包括逆向思维能力、良好记忆力、勇于怀疑和探索的精神、追求完美的品质等。比如逆向思维能力，开发人员通常从正面满足需求，很少去考虑不满足需求的部分；测试人员就要逆向思考，关注哪些开发人员没有考虑到的、不满足需求的内容。比如良好的记忆力，许多新出现的软件问题与之前曾经发现过的相差无几，好的测试者应有能力把以前遇到过的类似错误从记忆深处挖掘出来并举一反三，这一能力在测试过程中具有很高的价值。比如勇于怀疑和探索的精神，开发人员通常会尽量将所有的问题解释过去，软件测试人员需要听取开发人员的说明，但必须保持高度警惕和怀疑精神，直到自己分析清楚或亲自验证之后才能做出决定。软件测试人员不要害怕进入陌生环境，要勇于探索、勇于挑战，想方设法地找出隐藏在深处的错误。比如追求完美的品质，在测试过程中，软件测试人员常常会碰到转瞬即逝或者难以重现的软件故障，这时不要心存侥幸，而是要尽一切可能地去尝试和寻找，尽力接近目标，力求完美。

## 1.9　术语与缩略语

本节列出本文常用的术语与缩略语，它们的定义大都来自 GB/T 11457《信息技术　软件工程术语》等常用的标准规范。未列出的术语，请参见相关标准规范。

【软件　software】　与计算机系统的操作有关的计算机程序、规程和可能相关的文档。

【计算机程序　computer program】　计算机指令和数据定义的组合，它允许计算机硬件执行计算或控制功能。

【软件工程　software engineering】　应用计算机科学理论和技术以及工程管理原则和方法，按预算和进度，实现满足用户要求的软件产品的定义、开发、发布和维护的工程或进行研究的学科。

【软件配置项　software configuration item】　为配置管理设计的软件的集合，在配置管理过程中，作为单个实体对待。

【配置管理　configuration management】　应用技术的和管理的指导和监控方法以标识和说明配置项的功能和物理特征，控制这些特征的变更，记录和报告变更处理和实现状态并验证与规定的需求的遵循性。

【组件　component】　一个可被独立测试的最小软件单元。

【系统　system】　组织在一起实现一个特定功能或一组功能的一套组件。

【测试　testing】　a)在规定的条件下操作系统或组件、观察和记录结果并对系或部件的某些方面作评价的过程；b)分析软件项以检测在存在的和要求的条件之间的区别(隐错)以评价软件项的特征。

【测试用例　test case】　a)为具体的目标而开发的一组测试输入、执行条件和预料的结果；b)对于测试项，规定输入、预料的结果和一组执行条件的文档。

【白盒测试　white-box testing】　通过分析系统或组件的内部结构进行的测试，也称结构测试。

【黑盒测试　black-box testing】　忽略系统或组件的内部机制只集中于响应所选择的输入和执行条件产生的输出的一种测试，也成功能测试。

# 静态测试技术

软件测试技术可分为静态测试技术和动态测试技术。静态测试是与动态测试相对而言的，静态测试的最大特点，就是不需要执行被测软件，而动态测试需要执行一次或多次被测软件。静态测试与动态测试是互补的，通常组织更注重动态测试而不太重视静态测试，但静态测试往往能够以相对较低的代价发现被测软件存在的缺陷，包括需求文档或者其他相关文档的错误和二义性。特别是当动态测试成本较高时，尤其适用采用静态测试技术。

执行静态测试，需要软件需求规格说明、源程序代码以及其他诸如设计说明、用户手册等的相关文档，通常还需要使用一个或多个静态测试工具。如图 2-1 所示。

静态测试技术主要包括：文档审查、代码审查、静态分析、代码走查等。

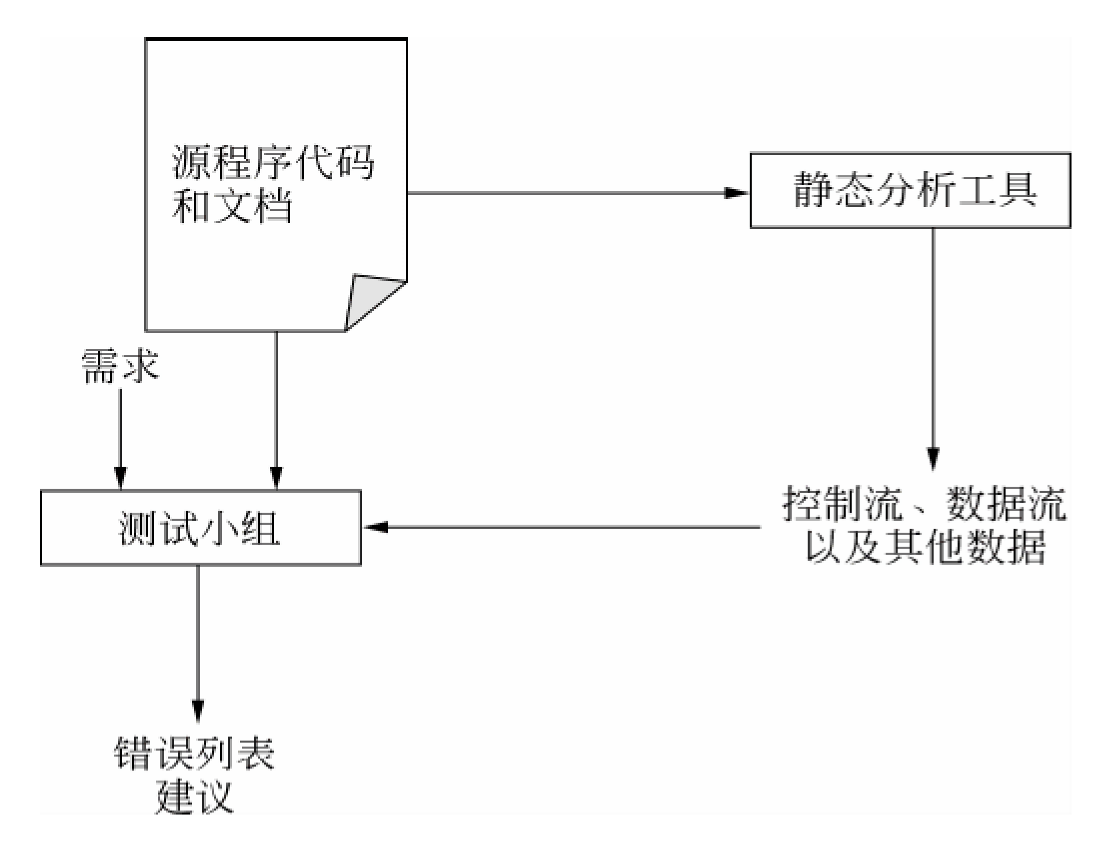



**图 2-1　静态测试的要素**

## 2.1　文档审查

文档审查是对软件文档进行静态审查的一项技术，审查对象一般包括软件需求规格说明、软件概要设计说明、软件详细设计说明、软件用户手册等各阶段文档，审查重点是文档的完整性、一致性和准确性。

文档审查应在审查前明确所使用的检查单。为适应不同类型文档的审查，需要使用不同的检查单，检查单的设计或采用应经过评审并得到委托方的确认。

## 2.1.1 实施要点

文档审查的实施要点主要有：

（1）确定需要进行文档审查的对象，一般仅审查技术文档，包括软件需求规格说明、软件设计文档、软件用户手册等；

（2）根据通用标准规范或委托方要求的工程规范对每份需要审查的文档制定文档审查单，并在测试策划评审时提交评审专家讨论通过，并得到委托方的确认；

（3）按照文档审查单进行文档审查，记录审查结果，报告发现的问题，需要时还应对更动后的文档进行回归审查，最后形成文档审查报告；

（4）文档审查应关注：文档格式是否符合规范要求，文档的描述是否明确、清晰，文档是否存在错误，文档之间是否一致等方面。

## 2.1.2 组织与流程

文档审查一般采用桌面检查、审查或正式评审的方式进行。

桌面检查是一种较早的人工检查方法，不需要召开会议，可看作是由单人进行的文档检查，成本较小，效率一般也较低。

审查是最重要的人工静态测试技术之一，通过把工作产品与事先定义的一组审查规则进行比较，在不运行程序的情况下进行测试，检测和发现软件工作产品中的问题和遗漏。审查是一个正式的技术活动，一般由软件产品开发人员和一个同行小组来执行，审查小组一般有 4～6 个成员，包括 1 个协调人或负责人、1 个记录人、1 个开发者代表（提供审查材料，实施问题的验证和确认）、1 个或多个同行专家。审查的基本步骤包括规划、准备、审查、返工和追查，如图 2-2 所示。在审查过程中需要执行的活动包括：

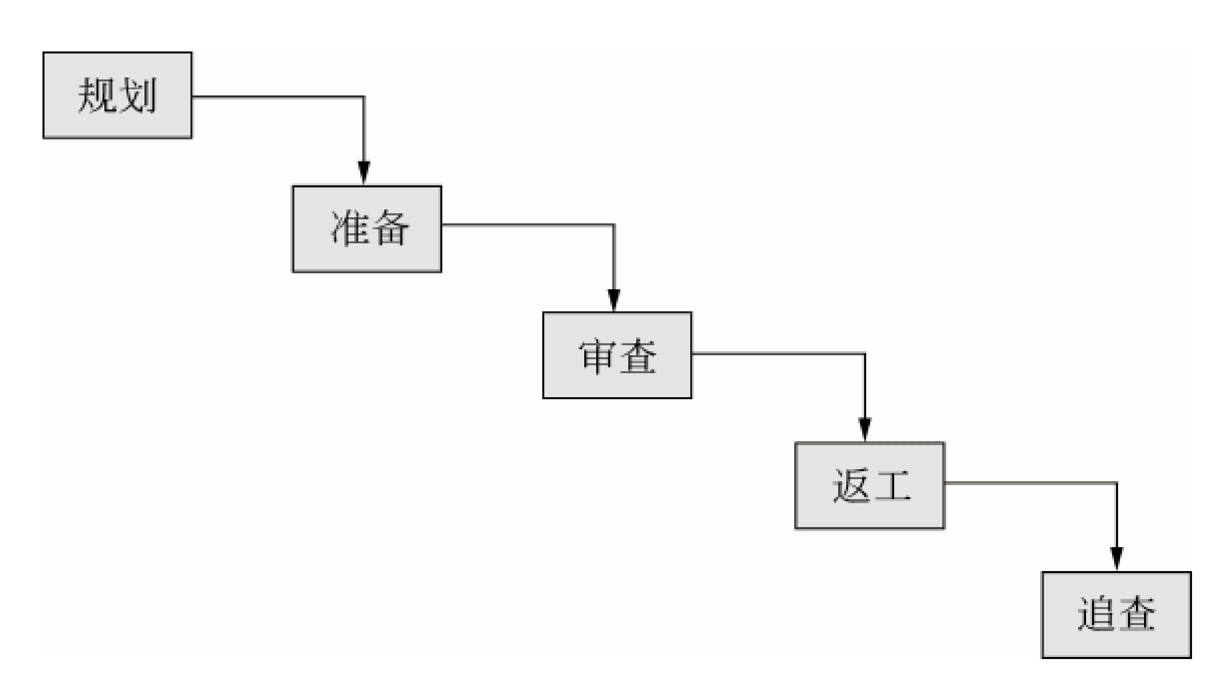



**图 2-2 审查的基本步骤**

（1）为审查小组成员提供准确的系统概述；

（2）事先把相关文档或产品分发给审查小组成员；

（3）在审查过程中发现错误并进行准确记录；

（4）在审查过程中提问并对问题进行追踪直到发现错误；

（5）对发现的错误进行验证和确认，以便于错误修复；

（6）根据结果决定是否需要再次审查。

正式评审被用于在开发进入下一阶段之前发现中间产品中存在的缺陷，通常在软件开发生存周期中每个阶段结束时实施，也可以在阶段中间出现严重问题的时候实施。正式技术评审始于 1976 年出现的 Fagan 代码审查技术，在 IBM 成功使用后被推广并演化开来。评审的目标是找出设计漏洞、缺陷或成本过高的部分，一般通过提交的相关文档或产品来尽早识别问题所在，并在评审前或评审时对问题逐一进行澄清。正式评审为在下个阶段开始前对项目进行有关变更提供了一次机会。关于评审的一般原则是：

（1）评审的对象是产品，并不针对开发人员；

（2）评审要有针对性，不能漫无目的地进行；

（3）要求事先充分准备，如果评审人没有准备好，应取消会议或重新安排时间；

（4）为每个被评审的文档或产品开发一个检查表；

（5）在评审时，只进行有限的争辩；

（6）阐明问题所在，但不要试图在会议中解决问题；

（7）翔实记录问题，列出问题、建议和解决该问题的负责人，并保留问题记录；

（8）将会议过程、发现的缺陷、向管理者提出的建议等进行文档化。

## 2.1.3　成果形式

文档审查的成果一般包括文档审查实施前的文档检查表和实施后的文档审查报告。

文档检查表根据通用标准规范或委托方要求的工程规范制定，需对每个要审查的文档制定一个。常见的软件需求规格说明、软件概要设计说明、软件详细设计说明、软件用户手册和软件系统设计说明的文档审查表，如表 2-1～表 2-5 所示。

文档审查报告的内容一般包括审查对象概述、审查时间、审查人员、审查地点、审查过程以及审查问题等。

**表 2-1　软件需求规格说明文档审查表**

| 序号 | 软件需求规格说明文档审查项 |
|---|---|
| 1 | 完整清晰地描述了引用文件，包括引用文档（文件）的文档号、标题、编写单位（或作者）和日期等 |
| 2 | 确切给出了所有在本文档中出现的专用术语和缩略语定义 |
| 3 | 以 CSCI 为单位，进行软件需求分析 |
| 4 | 采用了适合的软件需求分析方法 |
| 5 | 总体概述了每个 CSCI 应满足的功能需求和接口关系 |
| 6 | 完整、清晰、详细地描述由待开发软件实现的全部外部接口（包括接口的名称、标识、特性、通信协议、传递的信息、流量、时序等） |
| 7 | 完整、清晰、详细地描述由待开发软件实现的功能，包括业务规则、处理流程、数学模型、容错处理要求、异常处理要求等专业应用领域的全部要求 |

续表

| 序号 | 软件需求规格说明文档审查项 |
|---|---|
| 8 | 分别描述各个 CSCI 的性能需求 |
| 9 | 明确提出软件的安全性、可靠性、易用性、可移植性、维护性需求等其他要求 |
| 10 | 用名称和项目唯一标识号标识每个内部接口，描述在该接口上将要传递的信息的摘要 |
| 11 | 用名称和项目唯一标识号标识 CSCI 的数据元素，说明数据元素的测量单位、极限值/值域、精度、分辨率、来源/目的(对外部接口的数据元素，可引用详细描述该接口的接口需求规格说明或相关文档) |
| 12 | 指明各个 CSCI 的设计约束 |
| 13 | 详细说明在将开发完成了的 CSCI 安装到目标系统上时，为使其适应现场独特的条件和(或)系统环境的改变而提出的各种需求 |
| 14 | 描述运行环境要求，包括运行软件所需要的设备能力、软件运行所需要的支持软件环境 |
| 15 | 详细说明用于审查 CSCI 满足需求的方法，标识和描述专门用于合格性审查的工具、技术、过程、设施和验收限制等 |
| 16 | 详细说明要交付的 CSCI 介质的类型和特性 |
| 17 | 描述 CSCI 维护保障需求 |
| 18 | 描述本文档中的工程需求与“软件系统设计说明”和(或)“软件研制任务书”中的 CSCI 的需求的双向追踪关系 |
| 19 | 文档编制规范、内容完整、描述准确一致 |

**表 2-2　软件概要设计说明文档审查表**

| 序号 | 软件概要设计说明文档审查项 |
|---|---|
| 1 | 概述了 CSCI 在系统中的作用，描述了 CSCI 和系统中其他的配置项的相互关系 |
| 2 | 以 CSC 为实体进行了软件体系结构的设计 |
| 3 | 软件体系结构合理、优化、稳健 |
| 4 | 应对 CSC 之间的接口进行设计，用名称和项目唯一标识号标识每一个接口，并对与接口相关的数据元素、消息、优先级、通信协议等进行描述 |
| 5 | 为每个接口的数据元素建立数据元素表，说明数据元素的名称和唯一标识号、简要描述、来源/用户、测量单位、极限值/值域(若是常数，提供实际值)、精度或分辨率、计算或更新的频率或周期、数据元素执行的合法性检查、数据类型、数据表示/格式、数据元素的优先级等 |
| 6 | 规定每一个接口的优先级和通过该接口传递的每个消息的相对优先次序 |
| 7 | 描述接口通信协议，分小节给出协议的名称和通信规格细节，包括消息格式、错误控制和恢复过程、同步、流控制、数据传输率、周期还是非周期传送以及两次传输之间的最小时间间隔、路由/地址和命名约定、发送服务、状态/标识/通知单和其他报告特征以及安全保密等 |
| 8 | CSC 内存和处理时间分配合理(仅适用于“嵌入式软件”或“固件”) |
| 9 | 描述 CSCI 中各 CSC 的设计，将软件需求规格说明中定义的功能、性能等全部都分配到具体的软件部件，必要时，还应说明安全性分析和设计并标识关键模块的等级 |
| 10 | 用名称和项目唯一标识号标识 CSCI 中的全局数据结构和数据元素，建立数据元素表 |

续表

| 序号 | 软件概要设计说明文档审查项 |
| --- | --- |
| 11 | 用名称和项目唯一标识号标识被多个 CSC 或 CSU 共享的 CSCI 数据文件，描述数据文件的用途、文件的结构、文件的访问方法等 |
| 12 | 建立软件设计与软件需求的追踪表 |
| 13 | 文档编写规范、内容完整、描述准确一致 |

**表 2-3　软件详细设计说明文档审查表**

| 序号 | 软件详细设计说明文档审查项 |
| --- | --- |
| 1 | 概述了 CSCI 在系统中的作用，描述了 CSCI 和系统中其他的配置项的相互关系 |
| 2 | 以包或类的方式在软件体系结构范围内进行了逻辑层次分解，将软件需求规格说明中定义的功能、性能等全部进行了分配，分解的粒度合理，相关说明清晰 |
| 3 | 采用逻辑分解的元素描述有体系结构意义的用况，使体系结构设计与用况需求之间有紧密的关联 |
| 4 | 描述了系统的动态特征，对进程/重要线程的功能、生命周期和进程间的同步与协作有明确的说明 |
| 5 | 软件体系结构合理、优化、稳健 |
| 6 | 对每个标识的接口都设计有相应的接口类/包，规定每一个接口的优先级和通过该接口传递的每个消息的相对优先次序 |
| 7 | 描述接口和数据元素的来源/用户、测量单位、极限值/值域(若是常数，提供实际值)、精度或分辨率、计算或更新的频率或周期、数据元素执行的合法性检查、数据类型、数据表示/格式、数据元素的优先级等 |
| 8 | 进行安全性分析和设计并标识关键模块的等级 |
| 9 | 为完成需求的功能增加必要的包/类，使得层次分解的结果是一个完整的设计 |
| 10 | 实现视图描述 CSCI 的实现组成，每个构件分配了合适的需求功能，构件的表现形式(exe、dll 或 ocx 等)合理 |
| 11 | 部署视图描述 CSCI 的安装运行情况，能够对未来的运行景象形成明确概念 |
| 12 | 建立软件设计与软件需求的追踪表 |
| 13 | 采用的 UML 图形或其他图形描述正确、详略适当，有必要的文字说明 |
| 14 | 文档编写规范、内容完整、描述准确一致 |

**表 2-4　软件用户手册文档审查表**

| 序号 | 软件用户手册文档审查项 |
| --- | --- |
| 1 | 正确给出所有在本文档中出现的专用术语和缩略语的确切定义 |
| 2 | 准确描述软件安装过程，完整列出安装的有关媒体情况及使用方法 |
| 3 | 准确描述软件的各功能及操作说明，包括初始化、用户输入、输出、终止等信息 |
| 4 | 准确标识软件的所有出错告警信息、每个出错告警信息的含义和出现该错误告警信息时应采取的恢复动作等 |
| 5 | 文档编写规范、内容完整、描述准确一致 |

表 2-5　软件系统设计说明文档审查表

| 序号 | 软件系统设计说明文档审查项 |
|---|---|
| 1 | 总体概述了系统(或项目)的建设背景或改造背景,概述了系统的主要用途 |
| 2 | 引用文件完整准确,包括引用文档(文件)的文档号、标题、编写单位(或作者)和日期等 |
| 3 | 确切地给出所有在本文档中出现的专用术语和缩略语的定义 |
| 4 | 完整清晰描述软件系统的功能需求 |
| 5 | 完整清晰描述软件系统的性能需求 |
| 6 | 完整清晰描述软件系统的外部接口需求 |
| 7 | 完整清晰描述软件系统的适应性需求 |
| 8 | 完整清晰描述软件系统的安全性需求 |
| 9 | 完整清晰描述软件系统的操作需求 |
| 10 | 完整清晰描述软件系统的可靠性需求 |
| 11 | 清晰描述软件系统的运行环境 |
| 12 | 描述了系统的生产和部署阶段所需要的支持环境 |
| 13 | 以配置项为单位(包括软件配置项和(或)硬件配置项)设计了软件系统体系结构或系统体系结构 |
| 14 | 软件系统的体系结构合理、可行 |
| 15 | 用名称和项目唯一标识号标识每个 CSCI |
| 16 | 清晰、合理地为各个软件配置项分配了功能、性能 |
| 17 | 翔实设计了各个软件配置项与其他配置项(包括软件配置项、硬件配置项、固件配置项)之间的接口 |
| 18 | 进行了软件系统危险分析,合理确定软件配置项关键等级 |
| 19 | 合理分配了与每个 CSCI 相关的处理资源 |
| 20 | 追踪关系完整、清晰 |
| 21 | 文档编写规范、内容完整、描述准确、一致 |

# 2.2　代码审查

代码审查是对软件代码进行静态审查的一项技术,目的是检查代码和设计的一致性、代码执行标准的情况、代码逻辑表达的正确性、代码结构的合理性以及代码的规范性、可读性。代码审查应根据所使用的语言和编码规范确定审查所用的检查单,检查单的设计或采用经过评审并得到委托方的确认。

## 2.2.1　实施要点

代码审查的实施要点主要有以下 6 点。

(1) 对于代码执行标准的情况、代码逻辑表达的正确性、代码结构的合理性以及代码的可读性等,应明确规则检查标准,一般采用开发过程中遵循的标准,也可由测试方制定规则检查标准,规则检查标准需提交测试策划评审通过,得到委托方的确认。

(2) 尽可能选用相应代码的规则检查工具进行测试,对工具设置的检查规则应符合评审通过的规则。对于工具的检查结果,特别是问题部分,需要人工确认。

(3) 检查代码和设计的一致性需要阅读设计文档和代码,以检查代码实现是否与设计一致。

(4) 报告发现的问题,形成代码审查报告。

(5) 由于软件代码的复杂性,代码审查的通过标准不宜设为 100%满足;测试方可用百分比的方式提出建议通过标准,最终由委托方确定。

(6) 有条件时,在回归测试前,可对软件更改前后版本的代码进行比对。

## 2.2.2　组织与流程

代码审查采用自动化测试工具与人工确认相结合的方式进行,具体流程如下。

(1) 制定代码审查单。代码审查单应根据所使用的语言和编码规范制定,重点对代码执行标准的情况、代码逻辑表达的正确性、代码结构的合理性以及代码的可读性等进行检查,并需提交测试策划评审通过,得到委托方的确认。通常可采取在公认度高的通用检查单(比如 MISRA C++ :2008)的基础上根据具体情况剪裁的方式,制定所需要的代码审查单。

(2) 根据代码审查单,设定自动化测试工具的规则集,进行自动化代码规则检查。对于C/C++ 语言,经常使用的自动化测试工具包括 TestBed、CodeCast 等。自动化测试工具运行后,将生成自动化检查的结果,一般地,自动化检查结果中将包含大量的提示、警告和错误信息,其中可能含有相当大比重的虚警或误报信息。

(3) 采用人工方式对工具检查结果进行分析和确认。需对工具检查结果进行逐条分析,确认其指出的相应代码是否存在问题。必要时,可请软件开发人员对代码进行解释,协助确定问题。如果确实存在问题,应填写问题报告单。

(4) 采用人工方式,检查代码和设计的一致性。这需要阅读设计文档和代码,比较代码实现是否与设计一致,目前只能通过人工方式进行。同样地,如果存在问题,应填写问题报告单。在采用人工方式对工具检查结果进行确认以及检查代码和设计的一致性时,可以采用会议方式进行,应详细记录分析结果,特别是审查中发现的问题。应对问题的修改情况进行跟踪,必要时组织再次代码审查。

## 2.2.3　成果形式

代码审查的成果一般包括代码审查实施前的代码审查单和实施后的代码审查报告。

代码审查单应根据所使用的语言和编码规范制定,可采取在公认度高的通用检查单的基础上根据具体情况剪裁的方式进行。代码审查单示例如表 2-6 所示。

代码审查报告的内容一般包括审查对象概述、审查时间、审查人员、审查地点、审查过程、代码审查分析与统计结果(软件单元的规模、圈复杂度、扇入扇出数、源代码注释率、参数化率等静态特性)以及审查问题等。

**表 2-6 代码审查单示例**

| 序号 | 类 别 | 检 查 项 |
|---|---|---|
| 1 | 初始化和定义 | 只读存储器空白单元的处理是否合理 |
| 2 | | 随机存储器空白单元的处理是否合理 |
| 3 | | I/O 地址定义是否正确 |
| 4 | | 实际地址范围是多少？可寻址范围是多少？对实际地址范围以外的寻址是否进行了正确的处理 |
| 5 | | 变量是否唯一定义 |
| 6 | | 变量名称是否容易混淆 |
| 7 | 数据引用 | 是否引用了未经初始化的变量 |
| 8 | | 模块中间的数据关系是否符合约定 |
| 9 | 计算 | 数学模型的程序实现是否正确 |
| 10 | | 变量值是否超过有效范围 |
| 11 | | 对非法数据有无防范措施(如除法中除数为 0 等情况) |
| 12 | | 数据处理中是否存在累计误差 |
| 13 | | 是否对浮点数的上溢和下溢采取了合理的处理方式 |
| 14 | | 数据类型是否满足精度要求 |
| 15 | | 数组是否越界 |
| 16 | | 是否存在比较两个浮点数相等的运算 |
| 17 | 控制流 | 每个循环是否存在不终止的情况 |
| 18 | | 循环体是否存在循环次数不正确的可能，如是否存在迭代次数多 1 或少 1 的情况 |
| 19 | | 是否存在非穷举判断，如输入参数的期望值为“1”“2”或“3”，那么逻辑上是否可以判定该值非“1”、非“2”就必定是“3”，这种假设是否正确 |
| 20 | | 中断嵌套及现场保护是否正确 |
| 21 | | 条件跳转语句中的条件判断是否正确 |
| 22 | | 程序是否转错地方 |
| 23 | | 控制逻辑是否完整 |
| 24 | | 是否使用了 abort，exit 等跳转函数 |
| 25 | | 函数中是否存在多个出口 |
| 26 | | 循环中是否存在多个出口 |

续表

| 序号 | 类 别 | 检 查 项 |
|---|---|---|
| 27 | 多余物 | 用于增加程序的可测试性而引入的必要功能和特征是否经过验证,证明不会因此影响软件的可靠性和安全性 |
| 28 | | 是否存在不可能执行到的模块、分支、语句 |
| 29 | | 是否存在定义而未使用的变量及标号 |
| 30 | 安全可靠性设计 | 数据及标志有无防止瞬时干扰的措施,一般应采用“三比二”比对策略、定时刷新存储单元或回送比对后周期数据等措施 |
| 31 | | 重要数据的无用数据位是否采用了屏蔽措施 |
| 32 | | 对程序误跳转或跑飞是否采取了防范措施,如陷阱处理或路径判断 |
| 33 | | 重要信息的位模式是否避免采用仅使用一位的逻辑“1”和“0”表示,一般使用非全“0”或非全“1”的特定模式表示 |
| 34 | | 有无必要的容错措施 |
| 35 | 健壮性设计 | 对误操作是否有防范措施 |
| 36 | | 对于软件的重要功能或涉及系统安全性的功能,一旦硬件发生故障时,软件是否能继续在特定程序上维持其功能 |
| 37 | 格式 | 程序注释是否正确、有意义 |
| 38 | | 每个模块的入口处是否有头说明,包括功能、调用说明、入口说明、出口说明等 |
| 39 | | 程序模块的注释率是否符合要求 |
| 40 | 数据处理 | 缓冲区的使用是否合理 |
| 41 | | 数据处理流程是否高效、合理 |
| 42 | | 数据处理逻辑是否正确、合理 |
| 43 | | 数据处理是否通俗易懂 |
| 44 | | 数据处理方法是否简洁、高效、合理 |
| 45 | 其他 | 模块的规模,即代码行数是否符合要求 |
| 46 | | 模块的圈复杂度是否符合要求 |
| 47 | | 模块的扇入、扇出数是否符合要求 |
| 48 | | 模块的参数化率是否符合要求 |
| 49 | | 堆栈的处理是否合理,是否存在错误 |
| 50 | | 若使用了看门狗技术,其时间周期是否合理 |
| 51 | | 每个模块是否完成一个主要功能 |
| 52 | | 模块的入口、出口是否进行了现场保护 |
| 53 | | 全局变量的不恰当使用 |

## 2.3 静态分析

静态分析是一种对代码的机械性的和程序化的特性分析方法，主要目的是以图形的方式表现程序的内部结构，供测试人员对程序结构进行分析。静态分析的内容包括控制流分析、数据流分析、接口分析、表达式分析等，可根据需要进行裁剪，但一般至少应进行控制流分析和数据流分析。

### 2.3.1 实施要点

在静态分析中，测试人员通过使用静态分析测试工具分析程序源代码的系统结构、数据结构、内部控制逻辑等内部结构，生成函数调用关系图、控制流图、内部文件调用关系图、子程序表、宏和函数参数表等各种图形图表，可以清晰地展现被测软件的结构组成，并通过对这些图形图表的分析，帮助测试人员阅读和理解程序，检查软件是否存在缺陷或错误。

**1. 控制流分析**

20 世纪 70 年代以来，结构化程序的概念逐渐被人们接受，程序流程图(flowchart，又称框图)是人们最熟悉的一种程序控制结构的图形表示。程序流程图的框内常常标明处理要求和条件，而这些要求和条件在做路径分析时是不重要的。为了更加突出控制流的结构，人们对程序流程图进行了简化，称为控制流图(control-flowgraph)或程序图。程序图是有向图，是路径测试的基本依据。程序图中基本的控制结构对应的图形符号如图 2-3 所示。

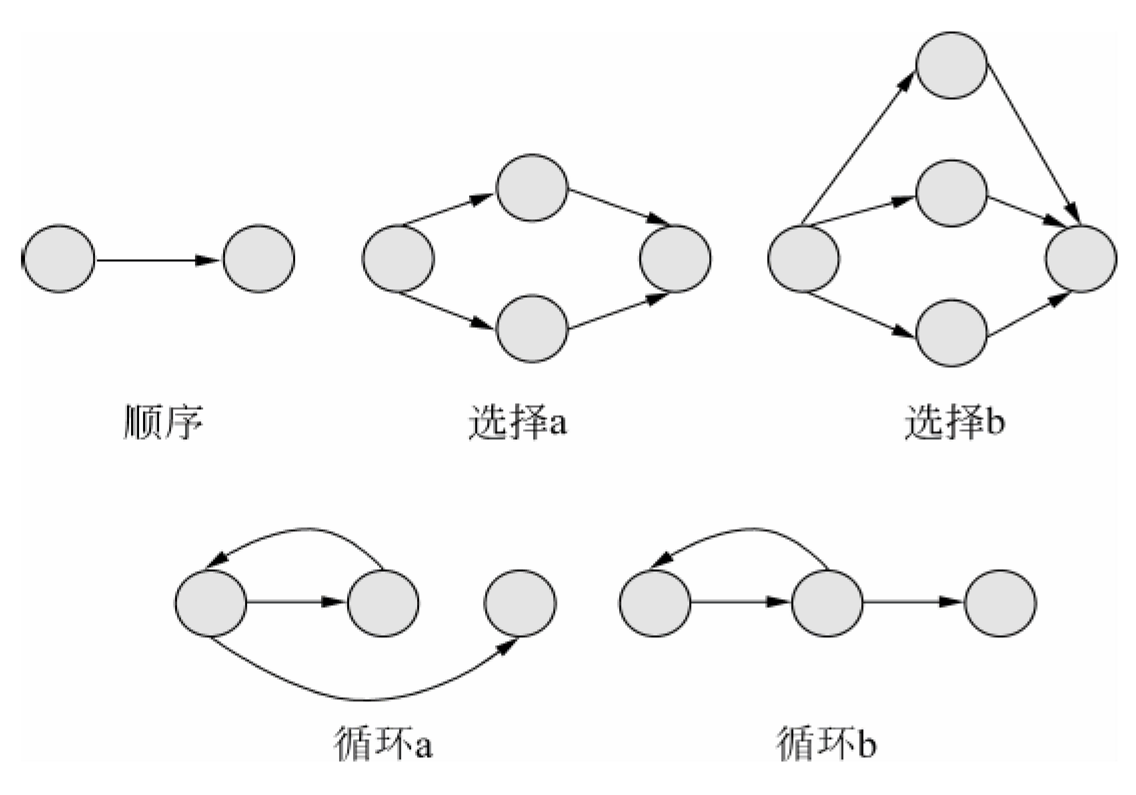



图 2-3 程序图的图形符号

控制流分析中常用的有函数调用关系图和函数控制流图。函数调用关系图通过树形方式展现软件各函数的调用关系，描述多个函数之间的关系，是从外部视角查看各函数；函数控制流图是由节点和边组成的有向图，节点表示一条或多条语句，边表示节点之间的控制走向，即语句的执行，是从函数内部考察控制关系，直观地反映函数的内部逻辑结构。

函数调用关系图的测试重点主要有：

(1) 函数之间的调用关系是否符合要求;

(2) 是否存在递归调用,递归调用一般对内存的消耗较大,对于不是必须的递归调用应尽量改为循环结构;

(3) 函数调用层次是否太深,过深的函数调用容易导致数据和信息传递的错误和遗漏,可通过适当增加单个函数的复杂度来改进;

(4) 是否存在孤立的函数,孤立函数意味着永远执行不到的场景或路径,为多余项。

函数控制流图的测试重点主要有:

(1) 是否存在多出口情况,多个程序出口意味着程序不是从一个统一的出口退出该变量空间,如果涉及指针赋值、空间分配等情况,一般容易导致空指针、内存未释放等缺陷;同时,每增加一个程序出口将使代码的圈复杂度增加 1,容易造成高圈复杂度的问题;

(2) 是否存在孤立的语句,孤立的语句意味着永远执行不到的路径,是明显的编程缺陷;

(3) 圈复杂度是否太大,一般地,圈复杂度不应大于 10,过高的圈复杂度将导致路径的大幅增加,容易引入缺陷,并带来测试难度和工作量的增加;

(4) 释放存在非结构化的设计,非结构化的设计经常导致程序的非正常执行结构,程序的可读性差,容易造成程序缺陷且在测试中不易被发现。

**2. 数据流分析**

数据流分析最初是随着编译系统有效目标码的生成而出现的,后来在软件测试中也得到成功应用,用于查找如引用未定义变量等程序错误或对未使用变量再次赋值等异常情况。

如果程序中某一语句执行时能改变某程序变量的值,则称此变量是被该语句定义的;如果某一语句的执行引用了内存中某程序变量的值,则说该语句引用此变量。

数据流分析考察变量定义和变量引用之间的路径,测试重点通常集中在定义/引用异常故障分析上,主要包括:

(1) 使用未定义的变量,如果一个变量在初始化前被使用,其当前值是未知的,可能会导致危险的后果;

(2) 变量已定义,但从未被使用,该类错误通常不会导致软件缺陷,但应对代码中的所有这种类型的问题进行检查和确认;

(3) 变量在使用之前被重复定义,变量在两次赋值之间未被使用,这种情况比较常见,大部分情况下也不会导致软件缺陷,但也应该进行检查和确认;

(4) 参数不匹配,指的是函数声明中的形参的变量类型与实参的变量类型不同,许多编译器对这种情况执行自动类型转换,但在某些情况下是危险的;

(5) 可疑类型转换,指的是为一个变量赋值的类型与变量本身的类型不一致,类型转换时两种类型看起来可能很相似,但赋值结果可能会导致信息丢失,如果无法避免,应使用显式的强制类型转换。

## 2.3.2　组织与流程

静态分析主要通过运行静态分析测试工具对程序代码进行自动化分析的方式开展,并需使用人工方式对测试工具生成的结果进行分析并得出结论。如果存在问题,应填写问题

报告单。

### 2.3.3 成果形式

静态分析的成果包括静态分析测试工具的原始结果，以及人工分析得出的结论。可以形成单独的静态分析报告，或者合并到其他静态测试（比如代码审查）的报告中。

## 2.4 代码走查

代码走查是对软件代码进行静态审查的一项技术，由测试人员组成小组，准备一批有代表性的测试用例，集体扮演计算机的角色，沿程序的逻辑，逐步运行测试用例，查找被测软件缺陷。

### 2.4.1 实施要点

代码审查的实施要点主要有：

（1）由测试人员集体阅读讨论程序，一般采用由开发人员逐行讲解代码、审查组集体讨论的方式进行；

（2）要准备一批有代表性的测试用例，要有确定的输入数据和输出结果，用人脑代替计算机执行测试用例、运行程序，并记录测试结果。

### 2.4.2 组织与流程

代码审查一般采用会议方式进行，需成立代码审查小组，代码审查小组一般有 5～8 个成员，包括 1 名协调人或负责人、1 名记录人、1 名开发者代表（提供审查材料，阅读代码，实施问题的验证和确认）、1 名或多名同行专家。审查的基本步骤包括规划、准备、审查、返工和追查，如图 2-2 所示。

在代码审查之前，开发者代表向审查小组负责人提供软件详细设计说明、程序清单、编码规范及相关文档等审查材料，并通过审查小组负责人把审查材料分发给小组成员，作为审查依据。审查小组负责人还应给每个成员分发已经过确认的代码审查检查单，也称缺陷检查表，列出以往编程中的常见错误，并对错误进行了分类。

审查小组成员在仔细阅读上述材料后，召开代码审查会议，进入正式的审查阶段。期间，开发者代表对程序逐句讲解，审查组其他成员听取讲解，提出自己的疑问，进而展开讨论，以确认是否存在错误或缺陷。

实践表明，编程人员在讲解自编程序的过程中，更易于发现原先未能发现的问题，同时审查小组成员的共同讨论，也有利于错误的暴露，发现更多的问题。

### 2.4.3　成果形式

代码走查的成果包括代码走查的测试用例以及代码走查结果。可以形成单独的代码走查报告，或者合并到其他静态测试(比如代码审查)的报告中。

## 2.5　静态测试技术分析

静态测试是一种不需要实际执行软件的测试方式，使用审查、走查、评审等方式进行。静态测试需要使用软件工具来保障覆盖性和自动化，也需要进行人工检查确认以保障测试效果。静态测试不需要执行代码，可以在软件开发过程的任何阶段开展，寻找软件中存在的错误和缺陷。

实际运行程序以测试软件功能和性能的测试，称为动态测试。动态测试与静态测试在测试效果上各有所长。相比于动态测试，静态测试的优点在于以下 6 点。

(1) 静态测试能够更早地发现软件问题。静态测试不需要执行代码，能够在需求分析、设计阶段甚至更早地开展。更早地发现问题，往往意味着能够节省大量的缺陷修正时间和金钱。

(2) 静态测试成本较低。静态测试采用审查、走查、评审等方式进行，通常不需要设计和执行大量的测试用例，发现缺陷的单位成本往往比动态测试低得多。

(3) 静态测试更加快速。代码审查、静态分析采用自动化测试工具与人工确认相结合的方式，文档审查和代码走查在执行前需要有较为充分的准备，但相对于动态测试，通常花费较短的时间。

(4) 静态测试发现的问题更容易定位。静态测试通过直接查看源代码或模拟执行代码进行测试，缺陷原因更容易分析，软件问题更容易定位。

(5) 静态测试能获得更高的覆盖率。由于只能在实际执行的代码中寻找缺陷，动态测试的语句覆盖率通常只能达到 60%至 70%，而静态测试往往能够在较短时间内达到 100%的覆盖。

(6) 静态测试可以发现更多类型的缺陷。静态测试能够在不同层次上发现编程缺陷，比如变量在初始化前使用、数组越界、不可达代码、循环中的无条件分支、参数类型或数目不匹配、未被调用的函数、空指针或指针类型错误等，而动态测试只能发现通过程序运行暴露出的缺陷。

静态测试也存在不少缺点，主要有以下 4 点。

(1) 静态测试工具经常会报告出大量的异常问题，使得判断哪些是真正的问题成为一项繁琐的工作。

(2) 静态测试无法看到代码之外需要分析的因素，比如：软件需求规格说明、操作系统、库文件等。

(3) 静态分析器对识别特定类型的代码问题还存在不足，比如函数指针、条件语句中的变量等。

（4）静态测试通常不作为一种详尽测试，只是检查代码/算法的健全程度以判断程序是否为详尽测试做好准备。

总而言之，静态测试侧重于文档及源代码的检查与优化，基本思想是不实际执行软件，直接查看源代码或模拟执行代码，目标是直接定位代码中的缺陷，提出结构设计优化和有关测试重点的意见建议。

CHAPTER 3 第3章

# 动态测试技术

与静态测试不同，动态测试需要首先设计测试用例，然后一次或多次运行被测软件，并通过分析软件运行结果与期望结果的差异，来分析被测软件是否满足要求。

前面已经介绍，软件测试有多种分类方法。从是否关注被测程序的内部结构和实现细节的角度，软件测试可分为白盒测试、黑盒测试，以及灰盒测试。

白盒测试利用程序设计的内部逻辑和控制结构生成测试用例，进行软件测试；黑盒测试方法主要通过分析规格说明中被测软件输入和输出的有关描述来设计测试用例，不需要了解被测软件的实现细节；灰盒测试是介于白盒测试和黑盒测试之间的一种测试方法，基于程序运行时的外部表现并结合程序内部逻辑结构来设计测试用例，采集程序外部输出和外部接口数据以及路径执行信息来衡量测试结果，对软件程序的外部需求及内部路径都进行检验。

## 3.1 白盒测试

### 3.1.1 概述

白盒测试(white-box testing)也称结构测试、逻辑驱动测试或基于程序的测试。根据GB/T 11457—2006，结构测试(structural testing)是“侧重于系统或部件内部机制的测试。类型包括分支测试、路径测试、语句测试”。白盒测试将测试形象地比喻成把程序放在一个透明的盒子里，如图3-1所示，测试人员了解被测程序的内部结构，利用程序的内部逻辑结

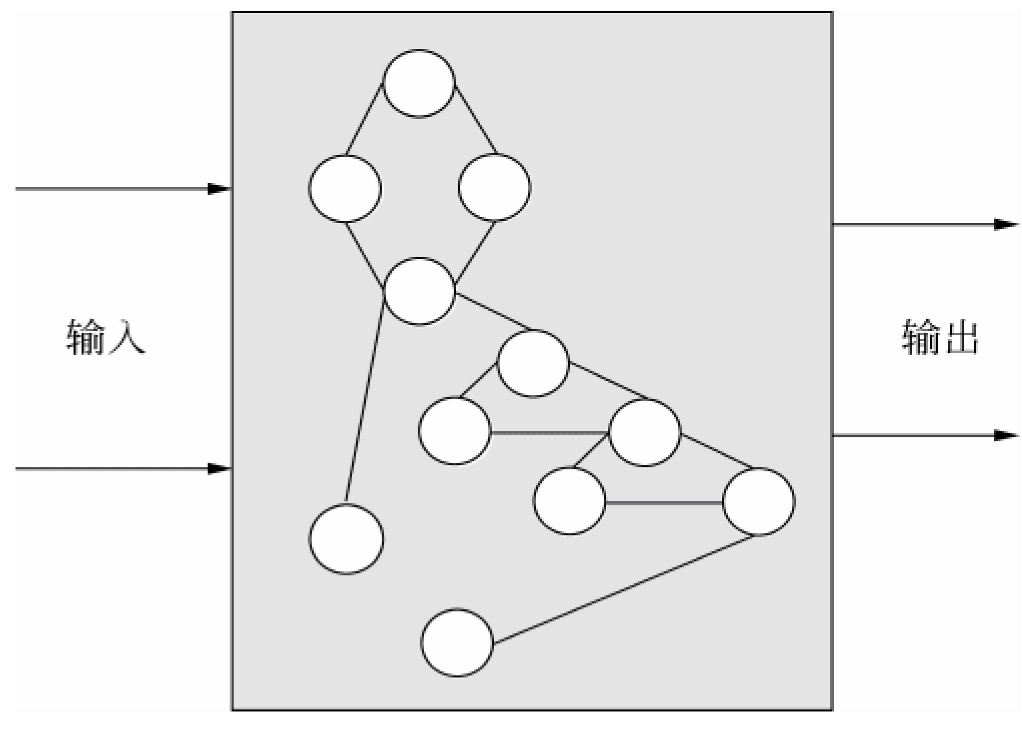



图3-1 白盒测试的基本原理图

构和相关信息，对程序的结构和路径进行测试。白盒测试是从程序设计者的角度进行的测试。

白盒测试的方法总体上可分为静态方法和动态方法两大类。静态方法是不实际执行程序而进行的测试，主要是检查程序代码或文档的表示和描述是否一致、符合要求以及有无冲突或歧义。文档审查、代码审查、静态分析、代码走查等都属于静态测试方法，已在第 2 章中进行了介绍。

动态测试的主要特点是当软件在真实的或模拟的环境中执行之前、之后及执行当中，对软件行为进行分析。动态测试时，软件在受控的环境下使用特定的期望结果进行正式运行，显示其在检查状态下是正确还是不正确。在本节后续小节中，将主要介绍动态白盒测试技术，包括基本路径测试、控制结构测试和其他的技术。

基本路径测试对程序设计复杂度进行合理度量，并以此为指导来定义一个基本路径集合。基本路径测试对所有独立路径进行测试，这些独立路径能组成程序的任意一条路径，因此能够满足分支测试的要求。基本路径测试通过对基本路径集生成测试用例，保证程序中的每条语句在测试中至少被执行一次。

控制结构测试是由基本路径测试演化而来的，对程序中语句或指令的执行顺序进行控制，其主要目标是选择测试用例以满足代码的各种覆盖准则。控制结构测试通常包括对判定的测试、对循环的测试、对数据流的测试等。

下面给出一段代码示例，本节后续小节将结合该代码开展相关介绍。

问题描述：NextDate 是一个函数，作用是根据输入的日期（年、月、日）计算后一天的日期。假设 NextDate 函数接收的输入值均为合法值，对输入值是否合法的判断在其他函数中完成，在此不再列出。

函数实现：列出 NextDate 函数的代码如下：

```
/// <summary>
/// 日期结构体
/// </summary>
public struct MyDate
{
    public int year;
    public int month;
    public int day;
}

/// <summary>
/// 计算下一天的日期
/// </summary>
/// <param name="date">当前日期</param>
/// <returns>下一天</returns>
MyDate NextDate(MyDate date)
{
    int year=date.year;
    int month=date.month;
    int day=date.day;
```

```
//计算当月的天数
int lastday=30;                              //定义月的天数(最后一天)
if (month==1 || month==3 || month==5 || month==7
    || month==8 || month==10 || month==12)
{//有 7 个月的天数为 31 天
    lastday=31;
}
else
{
    if (month==2)
    {//计算 2 月的天数,应为 28 天或 29 天(当闰年时)
        if ((year % 4==0 && year %100 !=0) || (year % 400==0))
        {//闰年
            lastday=29;
        }
        else
        {//非闰年
            lastday=28;
        }
    }
    else
    {//其他月份为 30 天
    }
}

MyDate tomorrow=new MyDate();  //下一天
tomorrow.year=year;
tomorrow.month=month;
tomorrow.day=day;

if (day==lastday)
{//如果当天是月末
    tomorrow.day=1;              //下一天是下个月的第一天
    if (month <12)
    {//未跨年
        tomorrow.month++;        //月份加 1
    }
    else
    {//跨年
        tomorrow.month=1;        //月份为下一年的第一个月(1 月)
        tomorrow.year++;         //年份加 1
    }
}
else
{//如果当天不是月末(是普通日期)
    tomorrow.day++;              //日期加 1
}

return tomorrow;
}
```

## 3.1.2 白盒测试基础

### 1. 控制流图和程序图

白盒测试依赖于程序的结构，需定义一种程序的表示方式。控制流图是一种刻画程序结构和逻辑流的方法，任何过程设计都可以转换为控制流图。

在控制流图中，线条和箭头表示流控制，称为边；圆圈表示一个或多个动作，称为节点；由边和节点围成的范围称为区域。如果一个节点包含判定条件，称为谓词节点。不同程序结构的控制流图表示如图 3-2 所示。

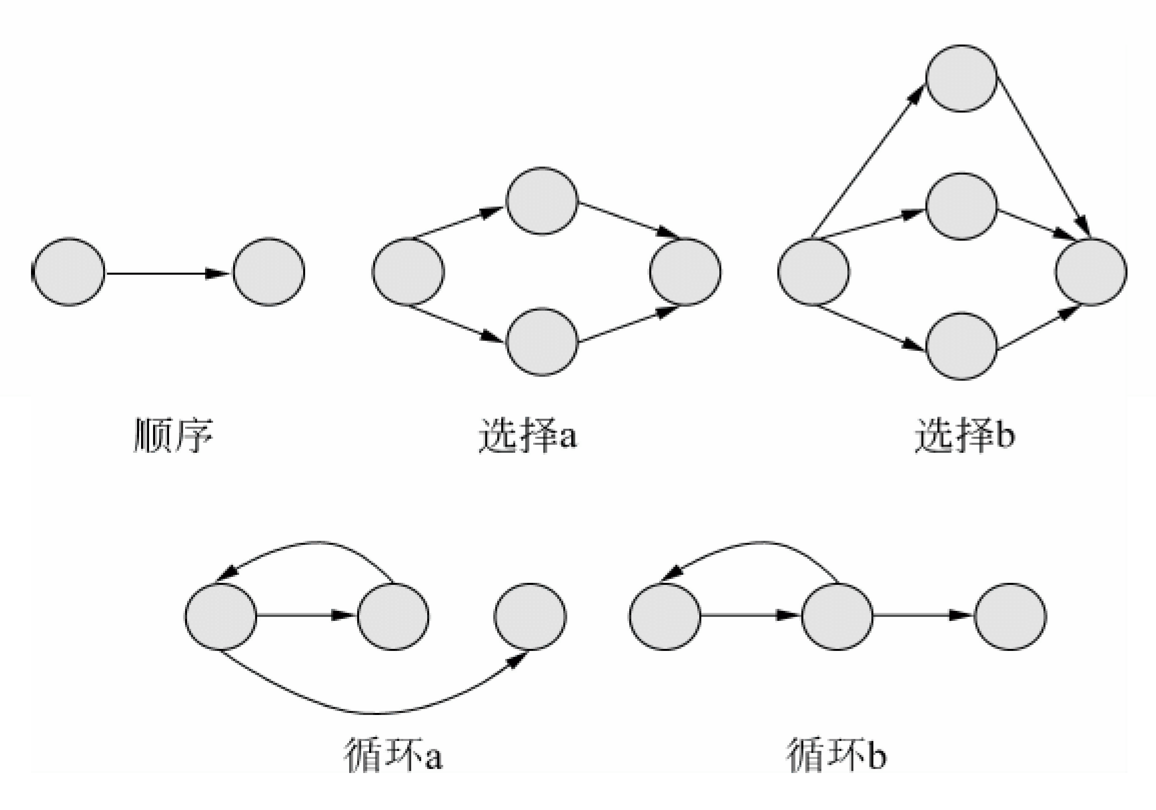



图 3-2　程序图的图形符号

程序图可以看作是压缩后的控制流图，也是一种特殊形式的有向图，上述 NextDate 函数的程序图如图 3-3 所示，图中数字对应源代码中的语句编号，K1、K2、K3 为语句段，其中 K1 表示语句 1～4，K2 表示语句 26～29，K3 表示语句 39～40。

程序图中每个节点代表一段语句片段(包含一条或多条语句)，每条有向边表示程序执行的走向。对一段程序源代码构造其对应的程序图，应遵循如下的压缩原则：

(1) 剔除注释语句，注释不参与实际程序执行，对程序结构不产生任何影响；

(2) 剔除数据变量的声明语句，此处的声明语句特指不进行初始化、只声明了变量类型的语句，进行初始化或赋值的语句不在此列；

(3) 所有连续的串行语句压缩为一个节点，即忽略子路径上经过的语句条数，无论某条子路径包含多少语句，只要不存在执行分支，一律压缩为一个节点，与变量无关；

(4) 所有循环次数压缩为一次循环，即忽略循环次数，无论某个循环结构将循环多少次，仅考虑执行循环体和不执行循环体这两种情况，与程序拓扑无关。

NextDate 的压缩后的程序图如图 3-4 所示，图中每个字母表示一段或一条语句，对应关系如下所示。在未特别指明的情况下，下文将以图 3-4 作为示例进行计算和介绍。

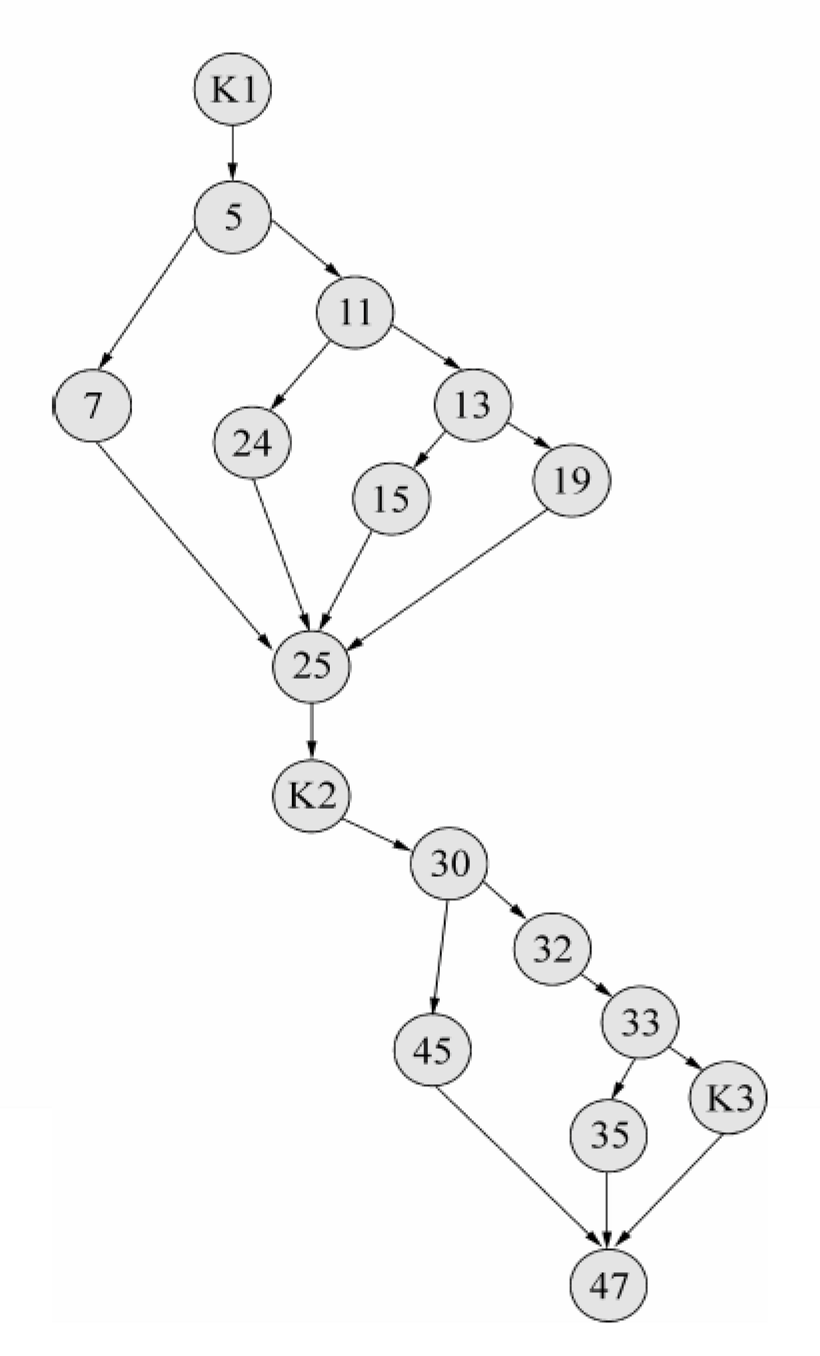



图 3-3　NextDate 的程序图

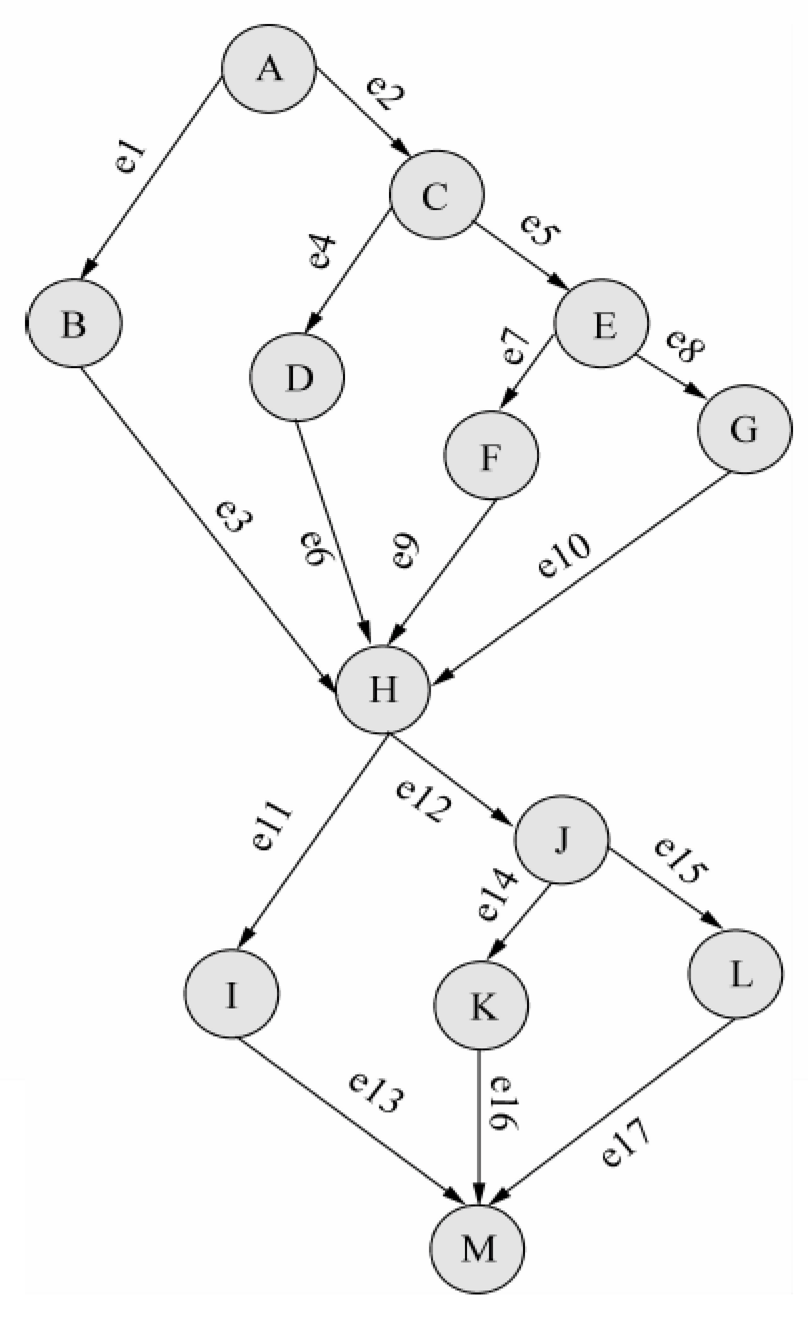



图 3-4　NextDate 的程序图(压缩后)

```
/// <summary>
/// 日期结构体
/// </summary>
public struct MyDate
{
    public int year;
    public int month;
    public int day;
}

/// <summary>
/// 计算下一天的日期
/// </summary>
/// <param name="date">当前日期</param>
/// <returns>下一天</returns>
MyDate NextDate(MyDate date)
{
    int year=date.year;
    int month=date.month;
    int day=date.day;
    //计算当月的天数                                         A
    int lastday=30;                  //定义月的天数(最后一天)
    if (month==1 || month==3 || month==5 || month==7
        || month==8 || month==10 || month==12)
```

```
    {//有 7 个月的天数为 31 天
        lastday=31;                                              B
    }
    else
    {                                                           }
        if (month==2)                                            C
        {//计算 2 月的天数,应为 28 天或 29 天(当闰年时)             }
            if ((year %4==0 && year %100 !=0) || (year %400==0))  } E
            {//闰年
                lastday=29;                                      F
            }
            else
            {//非闰年
                lastday=28;                                      G
            }
        }
        else
        {//其他月份为 30 天
        }                                                       } D
    }
    MyDate tomorrow=new MyDate();    //下一天                    }
    tomorrow.year=year;                                          }
    tomorrow.month=month;                                        } H
    tomorrow.day=day;                                            }
    if (day==lastday)                                            }
    {//如果当天是月末
        tomorrow.day=1;              //下一天是下个月的第一天      }
        if (month <12)                                           } J
        {//未跨年                                                 }
            tomorrow.month++;   //月份加 1                       K
        }
        else
        {//跨年
            tomorrow.month=1;   //月份为下一年的第一个月(1 月)     } L
            tomorrow.year++;    //年份加 1                       }
        }
    }
    else
    {//如果当天不是月末(是普通日期)
        tomorrow.day++;         //日期加 1                       }I
    }
    return tomorrow;
}                                                               }M
```

**2. 圈复杂度**

圈复杂度,又称环复杂度或 McCabe 复杂性度量,是一种对程序结构复杂度的度量模型,由 McCabe 于 1982 年提出,其基本思想是基于判定节点对程序图封闭圈数目造成的影响来衡量程序的复杂程度。根据圈复杂度,可以得出一个程序结构中独立路径的数目,以及确保每条语句被执行一次要生成的测试用例数目的上限。

计算一个程序的圈复杂度，有 3 种方式：观察法、公式法和谓词节点法。下面以图 3-4 为例分别介绍。

1）观察法

观察法指根据圈复杂度的定义，直观观察程序图将二维平面分隔成的封闭区域和开放区域的个数。对于图 3-4，其中封闭区域有 5 个：

区域 1，由节点 A、B、C、D、H 所围成；

区域 2，由节点 C、D、E、F、H 所围成；

区域 3，由节点 E、F、G、H 所围成；

区域 4，由节点 H、I、J、K、M 所围成；

区域 5，由节点 J、K、L、M 所围成；

另外还有 1 个外部的开发区域。因此，该程序图的圈复杂度为 6。

2）公式法

可利用程序图中边、节点和开放区域的数目，使用如下公式计算圈复杂度：

$$v(G) = e - n + 2p$$

其中，$v(G)$表示圈复杂度，$e$ 表示图中边的数目，$n$ 表示图中节点的数目，$p$ 表示图中未连接部分(即不封闭区域)的数目。

对于图 3-4，$v(G)=17-13+2\times 1=6$，该程序图的圈复杂度为 6。

3）谓词节点法

如果程序图中一个节点包含判定条件，称为谓词节点。可利用独立谓词节点的数目，使用如下公式计算圈复杂度：

$$v(G) = D + 1$$

其中，$D$ 表示图中独立谓词节点的数目。请注意，所谓的独立谓词节点数目不应简单地认为是程序图中判定节点的个数，事实上，分支大于 2 的判定节点可以被拆分成多个仅含两个分支的判定节点。因此，当程序中的判定节点均为两分支的判定时(包括循环节点)，可将每个判定节点视为一个独立谓词节点；而当判定节点为 $n$(大于 2)的分支时，该判定节点应视为($n-1$)个独立谓词节点。

对于图 3-4，节点 A、C、E、H、J 为独立谓词节点，一共有 5 个，所以，该程序图的圈复杂度为 6。

### 3. 图矩阵

图矩阵是一种有用的基本路径分析工具。图矩阵是一个方阵，其大小(行和列的数目)为流图中节点的数目，行或列表示流图的节点，矩阵中的元素表示流图的边。表 3-1 给出了图 3-4 的图矩阵，可以看到图矩阵就是一个表示流图的表格。

**表 3-1　NextDate 程序图的图矩阵**

| | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | | e1 | e2 | | | | | | | | | | |
| B | | | | | | | | e3 | | | | | |
| C | | | | e4 | e5 | | | | | | | | |

续表

| | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D | | | | | | | | e6 | | | | | |
| E | | | | | | e7 | e8 | | | | | | |
| F | | | | | | | | e9 | | | | | |
| G | | | | | | | | e10 | | | | | |
| H | | | | | | | | | e11 | e12 | | | |
| I | | | | | | | | | | | | | e13 |
| J | | | | | | | | | | | e14 | e15 | |
| K | | | | | | | | | | | | | e16 |
| L | | | | | | | | | | | | | e17 |
| M | | | | | | | | | | | | | |

给图矩阵中每个元素增加一个连接权值，图矩阵可被用于评估程序控制结构。权值可以表示不同的含义，比如：节点之间的连接存在或不存在，通常用“1”或“0”表示；一个连接(边)被执行的概率；在经历一个连接时所需的处理时间；在处理连接时所需的内存或其他资源等。作为示意，把表 3-1 中的边用“1”替代，表示该连接存在，给出表 3-1 的连接矩阵，如表 3-2 所示。连接矩阵可用于计算程序的圈复杂度。

**表 3-2 连接矩阵及圈复杂度**

| | A | B | C | D | E | F | G | H | I | J | K | L | M | 连接 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2－1＝1 |
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1－1＝0 |
| C | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2－1＝1 |
| D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1－1＝0 |
| E | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2－1＝1 |
| F | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1－1＝0 |
| G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1－1＝0 |
| H | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 2－1＝1 |
| I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1－1＝0 |
| J | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 2－1＝1 |
| K | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1－1＝0 |
| L | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1－1＝0 |
| M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 连接总数 | | | | | | | | | | | | | | 5 |
| 圈复杂度 | | | | | | | | | | | | | | 5＋1＝6 |

## 3.1.3　基本路径测试

### 1. 基本原理

从程序入口到程序出口之间存在许多可能的路径。对于判断语句，路径数目可能会加倍；对于 switch-case 语句，路径数目依赖于分支数目；对于循环语句，路径数目随着循环变量值增加而增大。总之，对于一段即使是比较简单的程序，要达到完全路径覆盖也是非常困难的。基本路径测试是一种较好的完全路径覆盖的变通方法。

基本路径测试的基本原理是：如果将全路径集合看作是一个向量空间，把从全路径集合中抽取的一组线性无关的独立路径看作是一组向量基，根据基于向量空间和向量基的理论可知，全路径集合中的所有路径可由这组独立路径的某种组合方式来遍历，因此，只需对这组独立路径进行了测试，就等价于对全路径进行了测试。这组独立路径就成为基路径或基本路径。基本路径测试的基本原理如图 3-5 所示。

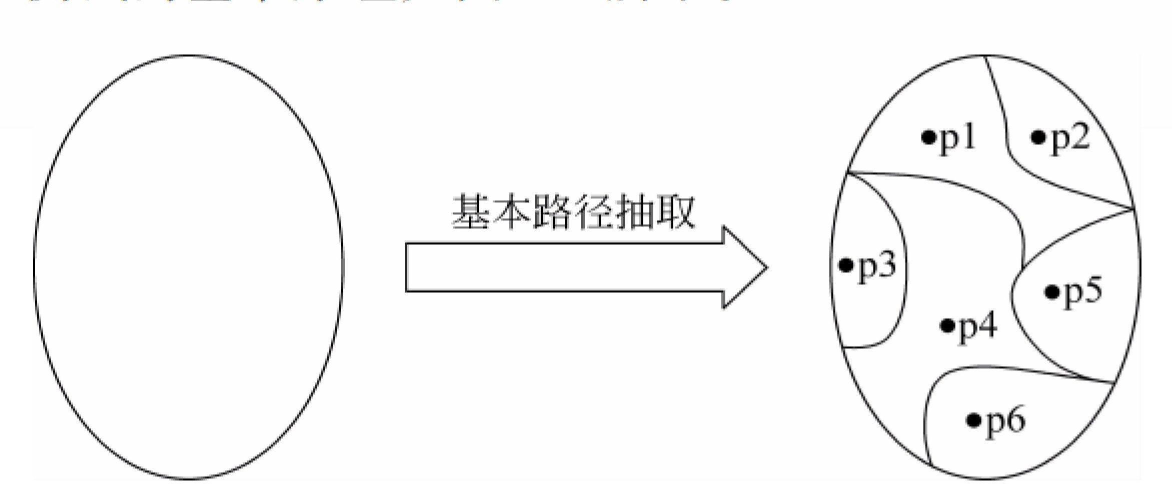



**图 3-5　基本路径测试的原理**

基本路径测试的目标是：

(1) 测试的完备性，即通过对独立路径的测试达到对所有路径的测试覆盖；

(2) 测试的无冗余性。每条路径都是独立的，设计的测试用例之间不存在冗余。

抽取的基本路径需满足如下要求：

(1) 任意两条基本路径线性无关；

(2) 所有基本路径的并是整个向量空间，即任意一条路径都可以转化为某一条或几条基本路径的组合遍历。

### 2. 测试设计

基本路径测试的主要步骤包括：以代码/设计为基础画出程序图；计算基本路径集合的规模；抽取基本路径；生成测试用例。结合 NextDate 程序进行介绍。

1) 画出程序图

根据 NextDate 程序代码(此处略，请见前面章节)，得到程序图，见图 3-4。

2) 计算基本路径集合的规模

基本路径的个数等于圈复杂度。对于 NextDate 程序，基于程序图可得圈复杂度是 6，其基本路径集合的规模也为 6。

3) 抽取基本路径

第一，确定主路径。从所有路径中，找到一条最复杂的路径作为主路径。所谓的最复杂体现在：

(1) 主路径应包含尽可能多的判定节点(包括条件判定和循环判定节点),判定节点越多,路径越复杂。

(2) 主路径应包含尽可能复杂的判定表达式,判定表达式包含的变量数量和“与”“或”关系越多,路径越复杂;当判定节点相同时,取判定表达式复杂的为主路径。

(3) 主路径应对应尽可能高的执行概率,每个判定节点取不同分支的概率并不相同,当不同路径包含相同数量的判定节点时,可根据一定规则来计算每条路径的执行概率(比如,所包含的所有判定分支执行概率的乘积),执行概率越高的路径越复杂。

(4) 主路径应包含尽可能多的语句,在相同执行概率的情况下,比较路径所包含的原始语句的数量,取语句数量多的为主路径。

第二,根据主路径抽取其他基本路径。

基于主路径,依次在该路径的每个判定节点处执行一个新的分支,构建一条新的基本路径,直至找到足够的基本路径数。当主路径上所有的判定节点处的每个分支都已覆盖,但仍不能达到指定数量的基本路径时,应查找程序中尚未完全覆盖的判定分支并构建基本路径。构建基本路径时,仍可按照判断表达式的复杂度、路径执行概率、路径包含语句数等原则进行。

对于 NextDate 程序,其基本路径集合为:

Path1:A-C-E-G-H-J-K-M(主路径);

Path2:A-B-H-J-K-M(在判定节点 A 处执行 e1 分支);

Path3:A-C-D-H-J-K-M(在判定节点 C 处执行 e4 分支);

Path4:A-C-E-F-H-J-K-M(在判定节点 E 处执行 e7 分支);

Path5:A-C-E-G-H-I-M(在判定节点 H 处执行 e11 分支);

Path6:A-C-E-G-H-J-L-M(在判定节点 J 处执行 e15 分支)。

4) 处理不可行路径

上面步骤完全是基于程序图进行的,而由于业务逻辑、程序缺陷等原因,得到的基本路径有可能是不可行路径,需要对不可行路径进行处理,通常有 2 种处理方法:

(1) 在不可行路径中,找出所有判定节点,进入另一个分支,直到得到符合业务逻辑的路径为止;

(2) 找到不可行路径,通过人工干预方式得到一条符合业务逻辑的路径。

对于 NextDate 程序,Path6 执行 A-C-E-G 语句,说明函数输入为 2 月(并且非闰年),从而不可能再执行判定节点 J 处的 e15 分支到达 L(跨年,只有 12 月才可以),因此,Path6 为不可行路径。

由于所有判定节点的每个分支都已覆盖,通过人工干预方式选定一条新路径作为 Path6:

Path6:A-B-H-I-M(发生几率最大)。

5) 设计测试用例

根据上面得到的基本路径设计对应的测试用例。

对于 NextDate 程序,针对基本路径 Path1~Path6,每条路径至少设计一个测试用例,得到测试用例集合如表 3-3 所示。

表 3-3　NextDate 的测试用例集

| 测试用例 ID | 输　　入 | 预 期 输 出 | 备　　注 |
|---|---|---|---|
| NextDate_01 | 2015-2-28 | 2015-3-1 | 对应 Path1 |
| NextDate_02 | 2015-7-31 | 2015-8-1 | 对应 Path2 |
| NextDate_03 | 2015-6-30 | 2015-7-1 | 对应 Path3 |
| NextDate_04 | 2012-2-29 | 2012-3-1 | 对应 Path4 |
| NextDate_05 | 2015-2-20 | 2015-2-21 | 对应 Path5 |
| NextDate_06 | 2015-1-15 | 2015-1-16 | 对应 Path6 |

**3. 基本路径测试技术分析**

基本路径测试既有优点，也有局限。

首先，在理论上基本路径测试保证了测试的完备性和无冗余性，并且可以大幅降低测试用例的数量。根据基本路径的抽取原则，如果所有要测试的路径是一个向量空间的话，基本路径集就是一组向量基，任意其他路径均可由基本路径的有限组合得到。

其次，不可行路径的存在可能会破坏基本路径测试的完备性和无冗余性。不可行路径可能是由于程序缺陷或不存在的业务逻辑等原因造成的，不可行路径的存在表明程序代码是不完备的。当然，根据程序实际应用场景或其他客观需求，不一定非要消除不可行路径，但至少指出了程序代码优化的方向。

第三，基本路径测试是基于程序图的，对串行语句长度和循环次数不敏感，能够保证覆盖到每个判定决策，并且能进一步保证对所有相互独立的判定决策结果进行测试，相比于判定覆盖指标更加健壮；但同时，基本路径测试不考虑每个分支的执行概率以及每条路径针对各数据变量的行为，与程序代码的实际情况是不吻合的，无法验证程序是否正确实现预期功能，无法发现程序违反设计规范之处，也很可能发现不了那些与数据相关的错误或用户操作相关的缺陷。

第四，基本路径可能是程序代码中执行概率很低的路径。当然也不能否认，执行概率低的路径往往是最有可能存在缺陷的路径。

最后，基本路径测试思想可用于任何动态模型中。在单元测试阶段，基本路径测试方法可主要用于对程序源代码的执行测试；在集成测试或系统测试阶段，基本路径测试方法可主要用于对业务流程、页面跳转等类似动态执行路径的测试。

## 3.1.4　控制结构测试

基本路径测试是简单而高效的，但有时还不能满足测试要求。由基本路径测试演化而来的控制结构测试不仅拓宽了测试覆盖准则，而且能够提高白盒测试的质量。

条件判定和循环是两类最重要的控制结构。

首先介绍对条件判定的测试，条件判定测试的覆盖准则有语句覆盖、判定覆盖、条件覆盖等多种，为便于说明，结合一段简单的代码进行介绍。

```
int Sample1(int a, int b, int c, int x)
{
    if ((a >0) && (b <0))
    {
        x=c;
    }
    if ((a==1) || (x >10))
    {
        x=x -10;
    }
    printf("a=%d,b=%d,c=%d,x=%d\n", a, b, c, x);
    return x;
}
```

代码对应的流程图如图 3-6 所示。该代码包含 4 个简单判定条件，分别是：(T1)$a>0$、(T2)$b<0$、(T3)$a==1$、(T4)$x>10$。该代码共有 4 条执行路径，分别是(P13)p1＋p3、(P24)p1＋p4、(P23)p2＋p3、(P24)p2＋p4。

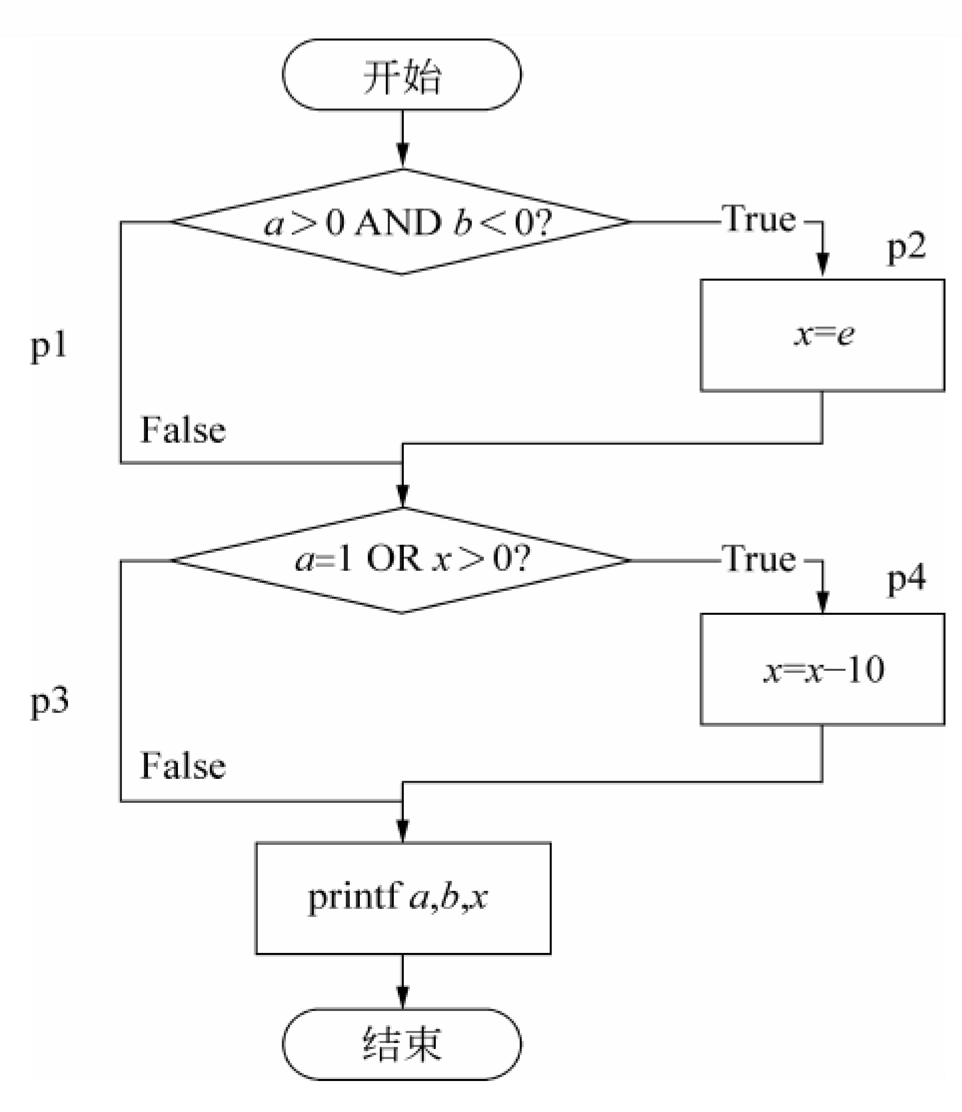



**图 3-6　函数 Sample1 的流程图**

需要注意的是，在上面代码中，没有 else 语句，两个判定节点均有隐式分支。这可能不影响代码的正确性，但不是良好的编程习惯，容易导致缺陷，这可从后续的测试设计中体现出来。

### 1. 语句覆盖

1) 含义

语句覆盖也称行覆盖、段覆盖或基本块覆盖，用于度量程序代码可执行语句被执行的比率。满足语句覆盖，指的是程序代码可执行语句至少被执行一遍，包括条件分支中包含的语句也要被执行到。即一个测试用例集合 $T=\{t_1, t_2, t_3, \cdots, t_k\}$满足语句覆盖准则，当且仅当对流程图中的任何节点 $v \in V$，存在测试用例 $t_i$ 及其相应执行路径 $P_i$，使得 $v \in P_i$。语句覆

盖等同于对图中所有节点的覆盖。

2）示例

为 Sample1 设计如表 3-4 所示的测试用例，均满足语句覆盖。

**表 3-4 语句覆盖的测试用例**

| ID | 输入 | | | | 预期输出 | 经过路径 | 语句覆盖 |
|---|---|---|---|---|---|---|---|
| | *a* | *b* | *c* | *x* | *x* | | |
| TC1-01 | 1 | −1 | 20 | 0 | 10 | p2＋p4 | 100％ |
| TC2-01 | 2 | −5 | 15 | 10 | 5 | p2＋p4 | 100％ |
| TC3-01 | 10 | −2 | 11 | −1 | 1 | p2＋p4 | 100％ |

由于 Sample1 的两个判定表达式的取假分支都不包含执行语句，为满足语句覆盖，设计的测试用例只需执行路径 p2＋p4 即可。语句覆盖对隐式分支无效。

3）分析

语句覆盖是所有覆盖准则中最弱的一个，既有优点也有缺点，主要包括：

（1）假设故障在整个代码中均匀分布，语句覆盖率能够反映故障发现的百分比；

（2）语句覆盖对控制结构不敏感，尤其对逻辑操作符完全不敏感；

（3）语句覆盖不区分循环语句，不能确定循环是否到达终止条件；

（4）在语句覆盖中，测试用例数目一般与分支点更相关；

（5）在保证所有语句被执行的情况下，很难对测试用例数目进行最小化。

**2. 判定覆盖**

1）含义

判定覆盖也叫分支覆盖，指的是程序代码中每个判定节点的取真和取假分支都被至少执行一次。判定覆盖等同于对图中所有边的覆盖。

2）示例

为 Sample1 设计如表 3-5 所示的测试用例集合 TC4、TC5，均满足语句覆盖和判定覆盖。

**表 3-5 判定覆盖的测试用例**

| ID | 输入 | | | | 预期输出 | 经过路径 | 语句覆盖 | 判定覆盖 |
|---|---|---|---|---|---|---|---|---|
| | *a* | *b* | *c* | *x* | *x* | | | |
| TC4-01 | 1 | −1 | 20 | 0 | 10 | p2＋p4 | 100％ | 100％ |
| TC4-02 | 0 | −5 | 15 | 1 | 1 | p1＋p3 | | |
| TC5-01 | 10 | −2 | 10 | −1 | 10 | p2＋p3 | 100％ | 100％ |
| TC5-02 | −1 | −2 | 5 | 10 | 10 | p1＋p4 | | |

3）分析

判断覆盖需覆盖到每一条执行边，测试重点转向判定节点，生成的测试用例的数目相对于语句覆盖有所增加，比语句覆盖具有更强的测试覆盖能力。满足判定覆盖的，一定满足百分之百语句覆盖。

但是，判定表达式多为复合判定，即由多个简单判定条件通过“与”“或”关系组合而成，而判定覆盖指标并未分析每个简单判定条件的取值情况，仍可能会导致遗漏部分缺陷。

### 3. 条件覆盖

1）含义

条件覆盖指的是，程序代码每个判定表达式中的每个条件的取真和取假情况至少被执行一次。

2）示例

对于 Sample1，为满足条件覆盖，就是要使得简单判定条件 T1～T4 的取真和取假分支至少执行一遍，设计测试用例集合 TC6 如表 3-6 所示。

**表 3-6 条件覆盖的测试用例 1**

| ID | 输入 | | | | 预期输出 | 简单判定条件 | | | | 经过路径 | 条件覆盖 | 判定覆盖 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *a* | *b* | *c* | *x* | *x* | T1 | T2 | T3 | T4 | | | |
| TC6-01 | 1 | 1 | 0 | 1 | －10 | T | F | T | F | p1＋p4 | 100% | 50% |
| TC6-02 | －1 | －1 | 0 | 20 | 10 | F | T | F | T | p1＋p4 | | |

可以看出，测试用例集合 TC6 满足条件覆盖，但却不满足判定覆盖。即满足条件覆盖的不一定能够满足判定覆盖。当然可以设计测试用例集合，使其及满足条件覆盖又满足判定覆盖，如表 3-7 所示。

**表 3-7 条件覆盖的测试用例 2**

| ID | 输入 | | | | 预期输出 | 简单判定条件 | | | | 经过路径 | 条件覆盖 | 判定覆盖 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *a* | *b* | *c* | *x* | *x* | T1 | T2 | T3 | T4 | | | |
| TC7-01 | 0 | 0 | 0 | 0 | 0 | F | F | F | F | p1＋p3 | 100% | 100% |
| TC7-02 | 1 | －1 | 20 | 0 | 10 | T | T | T | T | p2＋p4 | | |

3）分析

在介绍判定覆盖时，我们说判定覆盖指标只是要求每个判定节点的取真和取假分支都被至少执行一次，并未要求复合判定中的每个简单判定条件的每种取值至少执行一次；而条件覆盖指标有了此要求，即每个判定表达式中的每个条件的取真和取假情况至少被执行一次，但却没有要求每个判定节点的取真和取假分支都被至少执行一次。所以判定覆盖和条件覆盖是互不包含的。

### 4. 条件/判定覆盖

1）定义

为了同时满足对判定表达式所有分支的覆盖以及对判定表达式中每个简单判定条件的取值覆盖，引入条件/判定覆盖指标。条件/判定覆盖指标指的是，测试用例的设计应满足判定节点的取真和取假分支至少执行一次，并且每个简单判定条件的取真和取假情况也至少执行一次，即同时满足判定覆盖和条件覆盖。

2）示例

对于 Sample1，为满足条件/判定覆盖，设计如表 3-7 所示的一组测试用例即可。

3）分析

条件/判定覆盖对判定表达式的整体和局部同时进行覆盖，是一个较为完善的覆盖指标。不过，设计符合条件/判定覆盖指标要求的测试用例往往不是一件容易的事情。有时，我们可以通过修改程序结构，把复合判定表达式拆分成多个简单判定表达式，消除复合判定表达式中的“与”“或”关系，达到简化判定节点的目的；然后，只需简单使用判定覆盖就可以保证最终的测试用例同时满足条件覆盖。但是，也应该注意到，把复合判定表达式拆分成多个简单判定表达式时，往往会提高程序结构的复杂性，导致语句数目、路径数目的大幅增加。

**5. 条件组合覆盖**

1）定义

条件组合覆盖指的是程序代码中全部判定节点的每个简单判定条件的所有可能取值组合至少被执行一次。假设代码中有 2 个判定节点，且每个判定节点各包含 2 个简单条件，则此 4 个条件就有 $2^4=16$ 种组合，满足条件组合覆盖意味着要覆盖这 16 种组合。

2）示例

对于 Sample1，为满足条件组合覆盖，设计测试用例集合 TC8 如表 3-8 所示。4 个简单判定条件共产生 16 种组合，但有 4 种情况不可能发生，因为当简单条件 T1 取假时（即 $a\leqslant 0$），T3 只可能取假（即 $a\neq 1$），不可能取真。所以，最终产生 12 个测试用例。

**表 3-8　条件组合覆盖的测试用例**

| ID | 输入 | | | | 预期输出 | 简单判定条件 | | | | 经过路径 | 备注 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $a$ | $b$ | $c$ | $x$ | $x$ | T1 | T2 | T3 | T4 | | |
| TC8-01 | 1 | −1 | 11 | 0 | 1 | T | T | T | T | p2+p4 | |
| TC8-02 | 1 | −1 | 10 | 0 | 10 | T | T | T | F | p2+p4 | |
| TC8-03 | 2 | −1 | 11 | 0 | 1 | T | T | F | T | p2+p4 | |
| TC8-04 | 2 | −1 | 10 | 0 | 10 | T | T | F | F | p2+p3 | |
| TC8-05 | 1 | 1 | 0 | 11 | 1 | T | F | T | T | p1+p4 | |
| TC8-06 | 1 | 1 | 0 | 0 | 0 | T | F | T | F | p1+p4 | |
| TC8-07 | 2 | 1 | 0 | 11 | 1 | T | F | F | T | p1+p4 | |
| TC8-08 | 2 | 1 | 0 | 0 | 0 | T | F | F | F | p1+p3 | |
| TC8-09 | −1 | −1 | 0 | 11 | 1 | F | T | F | T | p1+p4 | |
| TC8-10 | −1 | −1 | 0 | 10 | 10 | F | T | F | F | p1+p3 | |
| TC8-11 | −1 | 1 | 0 | 11 | 1 | F | F | F | T | p1+p4 | |
| TC8-12 | −1 | 1 | 0 | 10 | 10 | F | F | F | F | p1+p3 | |
| | | | | | | F | T | T | T | | 不存在此情况 |

续表

| ID | 输入 | | | | 预期输出 | 简单判定条件 | | | | 经过路径 | 备注 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *a* | *b* | *c* | *x* | *x* | T1 | T2 | T3 | T4 | | |
| | | | | | | F | T | T | F | | 不存在此情况 |
| | | | | | | F | F | T | T | | 不存在此情况 |
| | | | | | | F | F | T | F | | 不存在此情况 |

3）分析

条件组合覆盖是以上指标中最完备的，满足条件组合覆盖就意味着一定同时满足判定覆盖和条件覆盖；但是，在判定表达式比较复杂且多个判定节点串联时，条件组合覆盖产生的测试用例的规模将会很大，实践中很可能难以实现。

**6. 修正的条件/判定覆盖**

1）定义

修正的条件/判定覆盖（MC/DC）指的是，在条件/判定覆盖的基础上，引入对简单条件的独立影响性的分析，减少条件/判定覆盖产生的冗余测试用例。

修正的条件/判定覆盖要求：判定中每个简单条件的所有可能结果至少出现一次，每个判定本身的所有可能结果也至少出现一次，每个入口点和出口点至少要执行一次，并且每个简单条件都能单独影响判定结果。

所谓独立影响是指在判定中其他条件不变的情况下，该条件的取值改变能引起整个判定结果的改变。比如在条件 B 必须为 True、条件 C 必须为 False 的情况下，判定结果因条件 A 的取值变化而变化，就称此时条件 A 独立影响判定结果。

修正的条件/判定覆盖的一般步骤是：

（1）列出所有的简单判定条件，构建真值表；

（2）对每个简单判定条件，找到能对整个判定结果产生独立影响的多组测试用例，称为独立影响对；

（3）抽取能体现所有简单判定条件独立影响性的最少独立影响对，就得到满足要求的测试用例集。

2）示例

```
int Sample2(int a, int b, int c, int d)
{
    if ((a==0||b==0) && (c==0||d==0))
    {
        printf("TRUE");
    }
else
    {
printf("FALSE");
    }
}
```

对于程序 Sample2，共有 1 个判定包含 4 个简单条件，设计如下表所示 5 个测试用例能满足 MC/DC 覆盖率。

| ID | *a* | *b* | *c* | *d* | MC/DC |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 100% |
| 2 | 1 | 0 | 1 | 0 | |
| 3 | 1 | 0 | 1 | 1 | |
| 4 | 1 | 0 | 0 | 1 | |
| 5 | 0 | 1 | 1 | 0 | |

3）分析

修正的条件/判定覆盖继承了条件组合覆盖的优点，并且有效控制了测试用例的数量，是一种较好的覆盖测试指标。它能发现的主要软件问题包括：操作符错误，比如“与”被误写为“或”；变量取反错误，即一个变量被误写为它的否定；表达式取反错误，即一个表达式被误写为它的否定。

在多个判定有关联的情况下，修正的条件/判定覆盖存在缺点。比如假设有两个判定 A＝B OR C 和 E＝A AND D，如果分别计算，则共有 4 个条件 B、C、A、D，{BCD}＝{TFT，FTF，FFT}就能达到 100%MC/DC 覆盖。但是，由于 A 的取值实际上是由 B 和 C 确定的，即上述两个判定等价于 E＝((B OR C) AND D)，而此时{BCD}＝{TFT，FTF，FFT}就不足以达到 100%MC/DC 覆盖，因为 C 和 D 都没有单独影响判定结果 E。因此，对于逻辑等价的两种情况，可能需要不同的测试用例集才能达到 100%MC/DC 覆盖。

**7. 对循环的测试**

循环是程序代码中除条件判定之外的另一类重要的控制结构。重复多次循环容易导致数据结构越界、内存泄漏等典型错误。因此，对循环的测试的基本思想是：重点关注循环的过程正确性，即在循环的边界和运行界限内对循环体的执行过程进行测试。循环结构可主要分为 3 类：简单循环、嵌套循环、非结构化循环。

1）简单循环

简单循环包括单个循环和以串联形式顺序执行的多个循环，如图 3-7 所示。这里针对单个循环进行介绍，串联循环可类似执行。

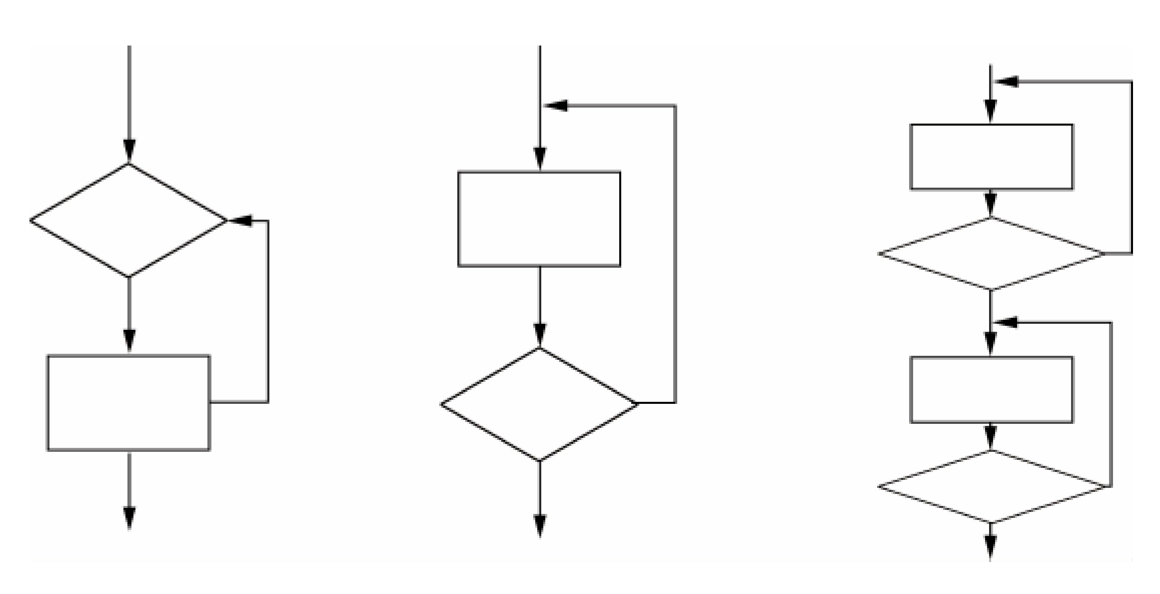


**图 3-7　简单循环**

对单个循环，假设最大循环次数为 $n$，在设计测试用例时应考虑如下情况：

(1) 循环 0 次，即不执行循环体；

(2) 循环 1 次，即执行一次循环；

(3) 循环 2 次；

(4) 循环 $m$ 次，$m$ 不大于 $n$，通常取 $n$ 的一半；

(5) 循环 $n-1$ 次；

(6) 循环 $n$ 次

(7) 循环 $n+1$ 次。

另外，在测试过程中，还应注意：

(1) 在循环初始化时，应考虑循环变量的初值是否设置正确，初值设置错误时，必将会影响循环总次数；

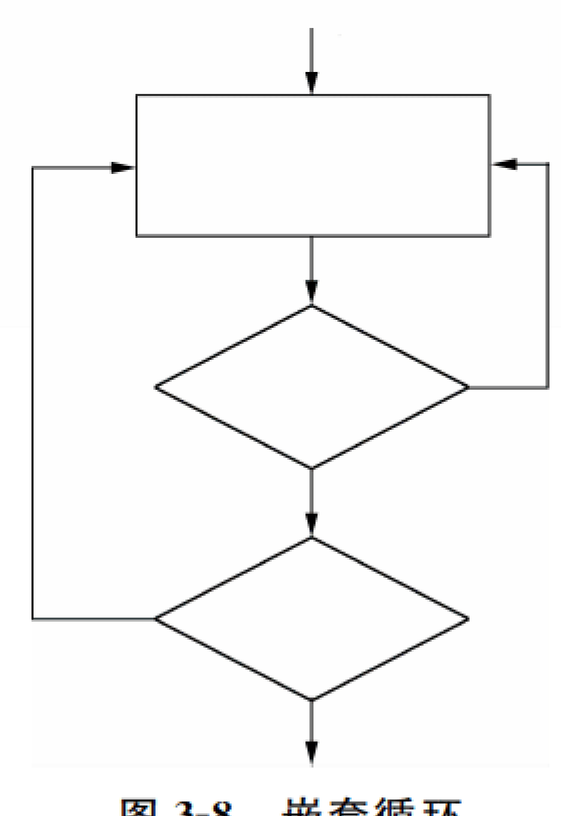


图 3-8　嵌套循环

(2) 在每次进行循环体内部后，应注意循环变量的增量是否正确，重要变量的取值是否按照预期规律变化，多次循环是否会导致误差累积，是否对内存造成压力，在 continue、break 等导致强制跳过部分语句的情况时是否存在错误等；

(3) 在循环终止时，循环变量的最大值是否正确，即循环的终止条件是否存在边界错误，退出循环的条件是否设置正确。

2) 嵌套循环

嵌套循环如图 3-8 所示。当循环节点以嵌套形式存在，且判定节点相互独立时，应按照由内至外的顺序，先测试最内层循环体，逐步外推，直至最外层的循环体。在测试每层循环体时，仍应根据单个循环的测试原则进行，并要考虑一下 4 种特殊情况：

(1) 内层最小循环次数且外层最小循环次数时；

(2) 内层最小循环次数且外层最大循环次数时；

(3) 内层最大循环次数且外层最小循环次数时；

(4) 内层最大循环次数且外层最大循环次数时。

3) 非结构化循环

非结构化循环如图 3-9 所示。非结构化的程序给测试带来很大难度，非结构化的循环通常无法测试，一般需重新设计成结构化的循环再进行测试。在结构化前必须执行测试的，可参照单个循环的测试原则，兼顾嵌套循环的组合情况开展。

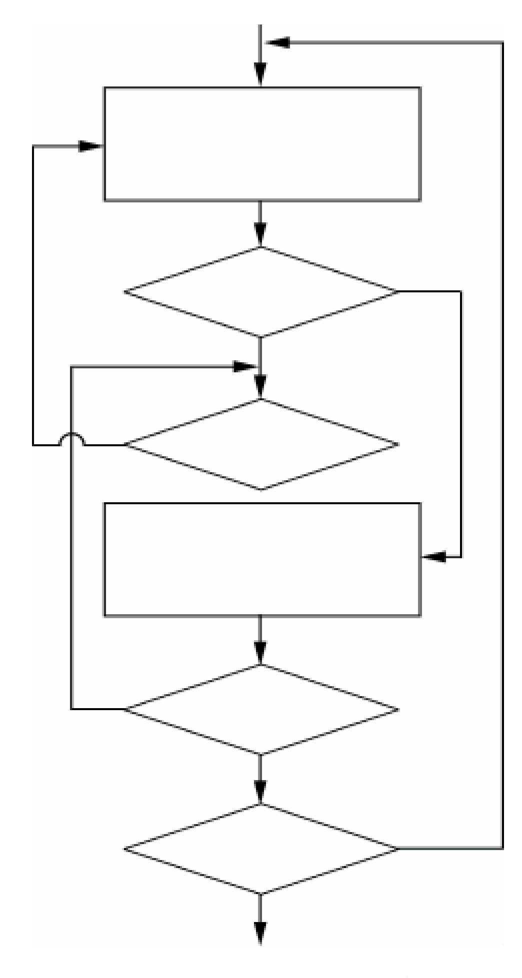


图 3-9　非结构化循环

**8. 控制结构测试技术分析**

对判定的测试主要是通过考察程序代码中复合判定表达式或构成复合判定表达式的各简单判定条件的所有取值情况，来保证判定表达式的正确性。常见的判定测试覆盖指标包括语句覆盖、判定覆盖、条件覆盖、条件/判定覆盖、条件组合覆盖以及修正的条件/判定覆盖，其中语句覆盖、判定覆盖、条件覆盖的使用是最为广泛的。

对循环的测试主要是在循环的边界和运行界限内对循环体的

执行过程进行测试。对于单个循环，需测试循环次数的边界以及循环过程中循环变量的初始化、递增和终止条件等；对于多个顺序执行的循环，可参照单个循环进行测试；对于嵌套的循环，除上述内容外，还应测试内外循环不同循环次数的边界组合；对于非结构化的循环，一般应先修改为结构化的。

控制结构测试的重点在于检查控制结构是否符合设计，并不考虑判定或循环体所涉及的变量所反映出的结果有何实际意义。事实上，这也是所有白盒测试方法的局限所在。

## 3.1.5　其他白盒测试技术

**1. 变异测试**

变异测试是一种基于故障的测试技术，可用于创建更有效的测试用例集。所谓的变异是指对源代码中的特定语句进行修改，然后检查测试用例能否发现该变异引发的错误。使用变异测试技术，首先为被测程序创建一组缺陷版本或变异体，然后设计测试用例去发现这些缺陷，而设计测试用例的目标就是能够区分原始程序与所有的变异体。

变异测试认为：如果所有简单故障都能被预测和删除，那么程序就已得到良好的测试，因为复杂故障是由简单故障组合而成的；给定一组合适的变异操作，如果一个测试集杀死了这些操作产生的变异，那么该测试集对于发现真实故障也会更加有效。

在变异测试中，需要创建程序的许多版本，每个版本通过变异植入一个故障，生成变异体；然后使用测试用例运行变异体，目标是其发生失效；如果引起失效，称该变异体被杀死，该测试用例被保存下来；直到所有的变异体被杀死，保留下来的所有测试用例组成测试用例集，用来对原始程序进行测试。变异测试的过程如图 3-10 所示。

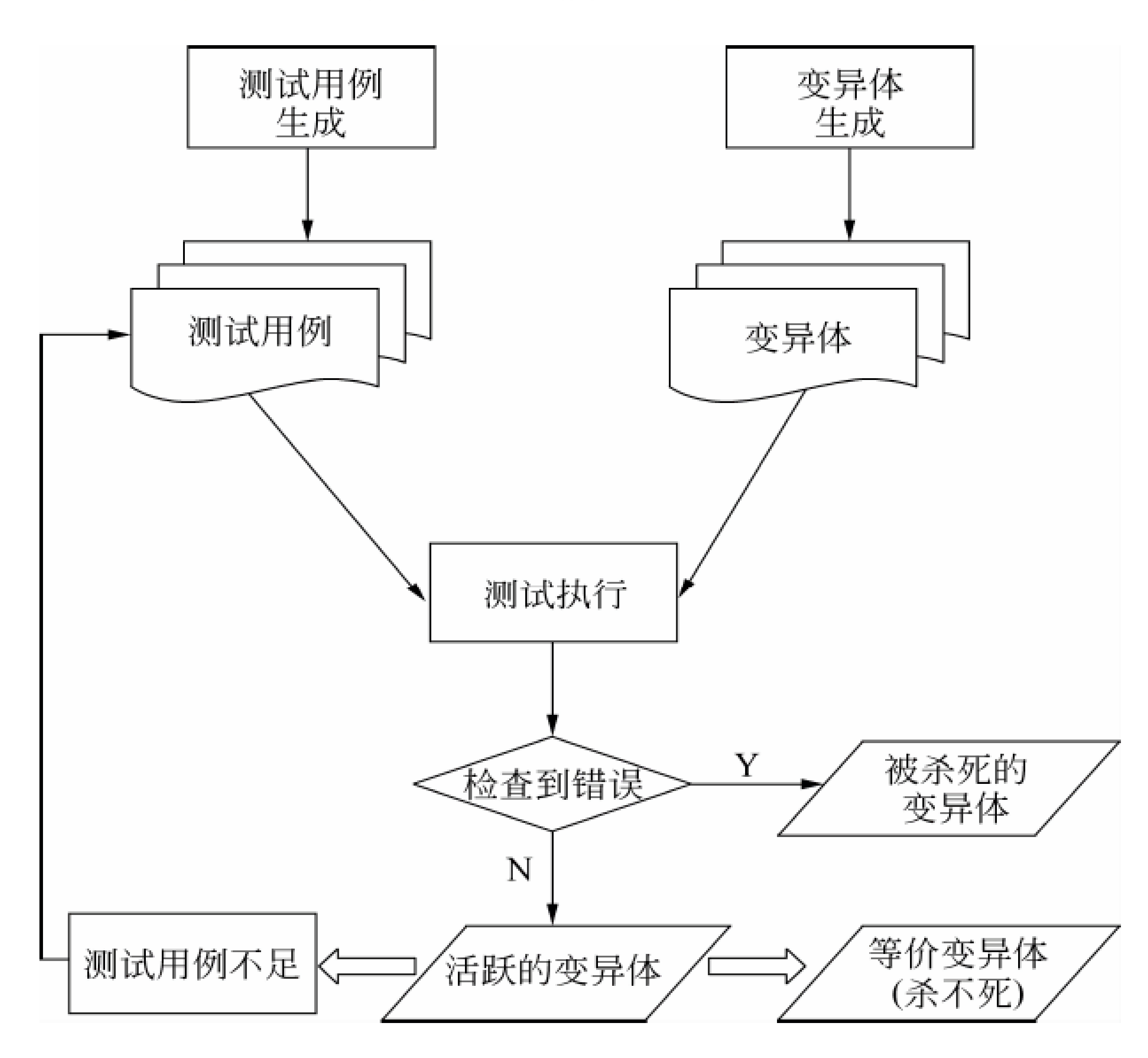



**图 3-10　变异测试的过程**

变异测试的优点在于：

(1) 能够发现代码中存在歧义及不明确的地方；

(2) 是一种容易被自动化执行的测试技术；

(3) 至少与数据流测试功能一样强大；

(4) 通过选择合适的变异操作，能执行更全面的测试；

(5) 最终能够得到缺陷更少和更可靠的程序代码。

变异测试通过对程序代码进行重写，以发现代码中可能存在的问题，其缺点主要有：

(1) 一组标准的变异操作会产生大量的变异体；

(2) 尽管变异体与最初程序不同，但它们可能具有相同的行为，而等价的变异体永远不能被杀死，识别它们很重要但非常困难；

(3) 有些顽固变异体可能不等价但也很难被杀死；

(4) 为了区分变异体和原始程序，可能需要执行大量的测试用例；

(5) 需测试大量的变异体程序，而每个变异体与原始程序的规模和复杂度都相当。

**2. 程序插桩**

程序插桩是借助于向被测程序中插入代码语句来实现测试目的的方法，是一种常用的有效测试方法。

程序插桩的基本原理是，在不破坏被测程序原有逻辑完整性的前提下，在程序的相应位置插入一些代码语句，称为探测器或探针，检测程序的运行特征或输出数据，并基于对这些数据的分析，揭示程序的内部行为和特征。比如，在程序中插入一些打印语句，使得程序执行时可输出我们关心的信息；或者在测试某一操作的处理时长性能指标时，在相应代码的前后分别插入语句，分别记录当前时刻，然后通过计算两个时刻的差获取该操作的时长。

在程序的特定位置插入语句以获取程序执行的动态数据，就像在刚研制成的机器的特定部位安装记录仪表，一方面可检验测试的结果数据，另一方面还可以了解程序的执行特性，实现探查或监控的功能，最终是为了把在程序执行过程中发生的重要事件记录下来，取得相应的测试或调试信息。例如，记录在程序执行过程中某些变量值的变化情况、变化范围等。

设计程序插桩时，需考虑的问题主要包括：

(1) 探测哪些信息；

(2) 在程序的什么部位设置探测点；

(3) 需要设置多少个探测点。

这些问题需要结合具体情况并针对程序的控制结构进行具体分析，一般地，应在遵循尽量设置较少探测点原则的基础上在以下部位设置探测语句。

(1) 程序块的第一个可执行语句之前；

(2) DO、DO WHILE、DO UNTIL 及 DO 终端语句之后；

(3) IF、ELSE IF、ELSE 语句之后；

(4) 输入输出语句之后；

(5) FOR、WHILE 语句的开始前和结束后。

**3. 域测试**

域测试是一种基于程序结构的测试方法。对于程序来说，每一条执行路径对应着一个输入域。可把程序执行路径错误分为 3 类：域错误、计算型错误和丢失路径错误。如果程

序的控制流有错误，那么对某一特定的输入可能执行的是一条错误路径，称为路径错误或域错误；如果对于特定输入执行的是正确路径，但由于赋值语句的错误导致输出结果不正确，称为计算型错误；如果由于程序中某处缺少判定谓词而引起路径执行错误，称为丢失路径错误。域测试主要是针对域错误进行的测试。

域测试的域是指程序的输入空间，域测试方法基于对输入空间的分析。程序的输入空间或称域，可分为不同的子空间，每一个子空间对应于不同的计算，而子空间的划分是由程序中分支语句的谓词决定的。输入空间的一个元素，根据其自身特征经过程序中某些特定语句后输出或结束（当然也可能存在无限循环而无出口的情况），形成一条执行路径。域测试正是在分析输入域的基础上选择合适的测试点后所开展的测试，基本步骤包括：

（1）根据各个分支谓词，给出子域分割图；

（2）在每个子域的边界处选取测试点；

（3）在子域内选取一些测试点；

（4）针对上述测试点进行测试。

域测试有两个主要缺点：一是为进行域测试，对程序提出的限制过多；二是当程序存在很多路径时，所需的测试点过多。

#### 4. 符号测试

符号测试可看作是对传统测试的一个扩充。符号测试的基本思想是不仅允许程序的输入是具体的数值数据，而且还可以是符号值。这里的符号值可以是基本符号变量值，也可以是这些符号变量值的一个表达式。从而在执行程序过程中，以符号计算代替了传统测试中的数值计算，得到的结果是符号公式或者符号谓词。

符号测试可看作是程序测试和程序验证的一个折中方法。一方面，它沿用了传统的程序测试方法，通过运行被测程序来验证其正确性；另一方面，由于一次符合测试的输入代表了一类传统测试的输入，实际上是证明了程序接收此类输入后的运行处理和输出结果。符号测试的关键是开发出比传统的编译器功能更强，能够处理符号运算的编译器和解释器。

目前符合测试还存在一些未得到圆满解决的问题，主要是以下 3 点。

（1）分支问题。当采用符号执行方法进行到某一分支点时，由于分支谓词是符号表达式，通常无法确定谓词的取值，也就不能决定分支的走向，需要测试人员进行干预或是执行树的方法进行下去。另外，如果程序中有循环，而循环次数又决定于输入变量，那就无法确定循环的次数。

（2）二义性问题。数据项的符号值可能是二义性的，这种情况通常出现在带有数组的程序中。比如下面的程序段：

```
x(i)=2+a;
x(j)=0;
b=x(i);
```

如果 $i=j$，则 $b=0$；否则 $b=2+a$。但如果使用符号值运算，无法知道 $i$ 是否等于 $j$，从而存在二义性。

（3）大程序问题。符号测试总是要处理符号表达式，随着符号执行的继续，一些变量的符号表达式会越来越庞大。特别是当符号执行树很大，分支点很多，路径条件本身变成一个

非常长的合取式。如果找不到化简的方法,将导致符号测试运行时间和空间的大幅增长,甚至带来难以克服的困难。

**5. 白盒测试技术分析**

白盒测试技术是软件测试的主要技术之一。白盒测试又叫结构测试或逻辑测试,利用程序设计的内部逻辑和控制结构生成测试用例,进行软件测试。白盒测试的优点是能够帮助软件测试人员提高对程序代码的覆盖率,从而发现代码中隐藏的问题,提高代码的质量。

白盒测试方法又可分为静态白盒测试方法和动态白盒测试方法。静态测试侧重于源代码检查和优化,基本思想是不设计测试用例,而是通过直接查看源代码或模拟执行代码的方式开展测试,目标是直接定位代码中的缺陷,提出结构设计优化的意见和有关测试重点的建议。动态测试侧重于关键程序结构的测试,基本思想是通过对导致程序结构复杂度的判定表达式、执行路径和循环结构等设计测试用例的方式开展测试,目标是达到某种程度的覆盖,从测试完备性和无冗余性的角度给予信心。

对使用各种白盒测试方法的总体策略,有如下 7 点建议。

(1) 优先进行静态白盒测试,特别是重要、核心的功能模块的相关代码,应定期组织严格评审,并不断更新缺陷检查表等测试标准或规范文档。

(2) 尽量使用测试工具完成代码结构和质量的相关分析和评估,对代码进行设计评审和优化。

(3) 采取先静态后动态的组合方式,即先利用工具或者人工手段进行静态结构分析、代码审查或代码走查,再进行基本路径测试、控制结构测试等覆盖率测试。

(4) 通过代码审查和动态测试的方式,对静态分析的结果做进一步确认,可使测试工作更加有效。

(5) 覆盖率测试是白盒测试的重点,要选择合理的覆盖率指标,一般可先使用基路径测试方法,然后对重点单元或模块,使用多种覆盖率标准衡量对代码的覆盖率。

(6) 设计测试用例时,注意结合边界抽取测试数据,包括判定表达式的边界、循环次数的边界、变量取值范围的边界等。

(7) 在不同测试阶段,测试的侧重点有所不同:在单元测试阶段,以代码审查、逻辑覆盖为主;在集成测试阶段,需要增加静态分析、静态质量度量;在系统测试阶段,应基于黑盒测试的结果和需要,采取相应的白盒测试方法,可借鉴基本路径测试的思想设计高层测试用例,提高测试的覆盖性,降低测试的冗余。

## 3.2 黑盒测试

### 3.2.1 概述

黑盒测试(black-box testing)也称功能测试、数据驱动测试或基于规格说明的测试。根据 GB/T 11457—2006,功能测试(functional testing)是"忽略系统或部件的内部机制只集中于响应所选择的输入和执行条件的输出的一种测试"。黑盒测试把被测软件看作是一个打不开的黑盒子,如图 3-11 所示,测试人员关注软件行为和功能,不关心软件内部结构,根据

软件规格说明来发现程序中的错误。黑盒测试是从程序使用者或用户的角度进行的测试。




**图 3-11　黑盒测试技术示意图**

黑盒测试主要发现以下类型的错误：

(1) 基于规格说明的功能错误，包括不正确的和遗漏的功能；

(2) 基于规格说明的软件或系统行为错误；

(3) 基于规格说明的性能错误；

(4) 面向用户的使用错误；

(5) 黑盒接口错误，包括接口数据结构或外部访问错误等；

(6) 初始化或终止错误。

黑盒测试在设计测试用例时只对输入和输出进行考虑。由于对所有可能的输入进行穷尽测试是不现实的，因此测试人员的目标是在可用资源约束下生成测试用例，尽可能多地发现软件存在的问题。常用的黑盒测试方法包括等价类划分、边界值分析、错误推测、基于场景测试、因果图与决策表法等，将在后面章节逐一介绍。

为了方便，将继续结合前面白盒测试所用的 NextDate 函数进行介绍，代码不再重复列出。NextDate 函数在这里可看作是一个小软件，其功能是根据输入的日期(年、月、日)计算并输出后一天的日期。这里仍然假设 NextDate 函数接收的输入值均为合法值，对输入值是否合法的判断在其他函数中完成。

## 3.2.2　等价类划分

### 1. 概念定义

等价类划分法是一种典型的黑盒测试方法。软件测试的基本原则之一就是不可能做到穷尽测试，为了满足测试的完备性和无冗余性的目标，人们首先提出了等价类划分法。

等价类划分法就是通过等价划分的方式将数据分类，然后从每个类别中抽取典型数据实施测试。以被测软件的输入域为例，若要满足测试用例对整个有效输入域和无效输入域的完全覆盖，并且无冗余，可在理论上找到一种划分方式，将该输入域划分为一些子集(称为等价类)，满足：

(1) 每个子集内所有数据等价，即被测软件对该子集中每个数据的处理方式相同(保证覆盖)；

(2) 各子集互不相交，即输入域中的某个数据唯一隶属于某个子集(保证无冗余)；

(3) 所有子集的并集是整个输入域(保证完备)。

由此，若从每个子集中随便抽取一个(且仅一个)数据，所构成的数据集合可满足测试的完备性和无冗余性，从而将难以穷举的输入域数据减少为少量典型数据的集合，降低测试工作量。

实际上，为了避免等价类测试的漏洞，不仅需要对被测软件的输入域进行等价类划分，

还常常需要对输出域进行等价类划分。

### 2. 原理分析

所谓等价类是指输入域(和(或)输出域)的一组互不相交的子集,该组子集的并集是整个输入域,因此等价类划分法设计的测试用例具备完备性和无冗余性。由于等价类是由等价关系决定的,等价类中的元素具有一些共同特点。如果用等价类中的一个元素作为测试数据不能发现程序缺陷,那么使用集合中的其他元素也不能发现该缺陷,即对于揭露程序缺陷来说,等价类中的每个元素是等价的。应该将测试用例分布到各个等价类中。

软件不能只接受有效合理的数据,还要能处理无效或非法的数据,这样才能具有较高的可靠性。因此,在等价类划分时,应区分两种不同的情况。

(1) 有效等价类,即符合规格说明的,有意义的、合理的数据构成的集合。利用有效等价类,能够检验程序是否实现了规格说明中所规定的要求。在具体问题中,有效等价类可以是一个,但一般是多个。

(2) 无效等价类,即不符合规格说明的,不合理或非法的数据构成的集合。利用无效等价类,能够检查软件功能和性能的实现是否有不符合规格说明的地方。在具体问题中,无效等价类可以是一个,也可能有多个。

### 3. 测试设计

等价类划分可基于一种非常简便的思路:首先,将某个输入条件的所有可能取值划分为一个有效等价类,其余取值划分为一个无效等价类;然后,针对有效等价类,通过不断施加规则,将满足规则和不满足规则的数据划分为不同的有效等价类,不断重复,直至无法继续划分为止,最后得到的每个有效等价类代表了被测程序的一种特殊处理方式;最后,对于无效等价类,也采取相同方式进行划分,不过一般没有必要对无效等价类进行过多的等价划分。

就某个输入条件而言,一般地,等价类划分可参照如下原则:

(1) 若输入条件规定了取值范围,且取值范围上限、下限之间的数据是有意义的数据,则取值范围内的数据构成一个等价类,小于下限或大于上限的所有数据分别构成两个无效等价类;

(2) 若输入条件规定了“必须如何”的条件,在满足必须条件的数据构成一个有效等级类,其他数据构成一个无效等价类;

(3) 若输入条件是一个布尔量,则取真值的数据构成一个有效等价类,取假值的数据构成一个无效等价类;

(4) 若输入条件是一个枚举量,即规定了输入数据的一组合法值,并且程序要对每个输入值分别进行处理,则可为每一个输入值划分一个有效等价类,此外还要对这组合法值之外的值确立一个无效等价类。

### 4. 实例演示

以 NextDate 为例。假设允许输入的最小日期为 1900 年 1 月 1 日,最大日期为 2099 年 12 月 31 日,为了叙述方便,分别记为 MinDate 和 MaxDate。

首先,根据有效输入域的最小值和最大值得到:

有效等价类:Dt={日期|从 MinDate 开始到 MaxDate 截止的所有日期};

无效等价类:NDt1={日期|MinDate 之前的所有日期},NDt2={日期|MaxDate 之后

的所有日期}。

然后,通过不断施加规则,将有效等价类不断划分下去,如图 3-12 所示。

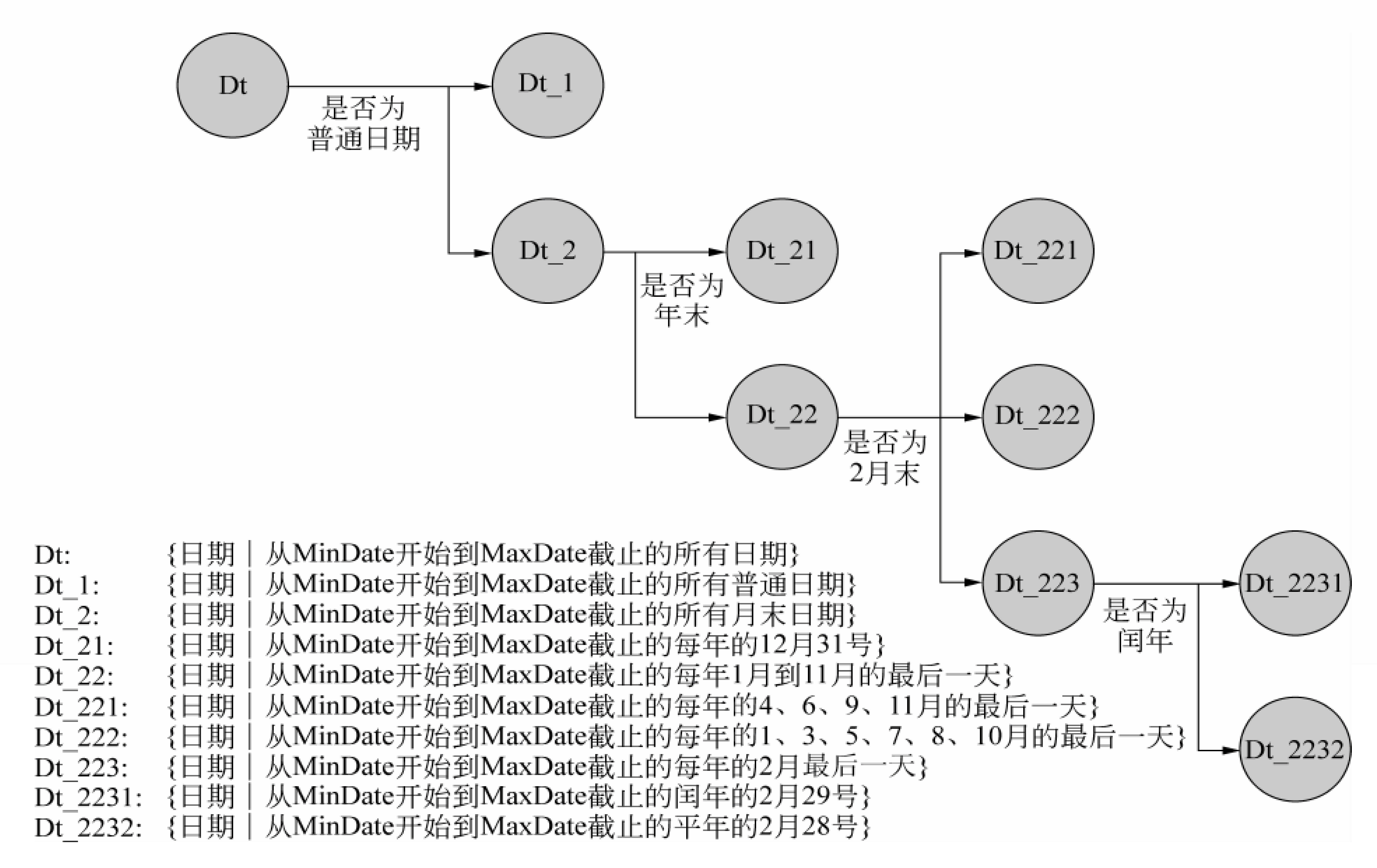



**图 3-12 NextDate 的有效等价类划分示意图**

最终将有效输入划分为 Dt_1、Dt_21、Dt_221、Dt_222、Dt_2231、Dt_2232 六个等价类,对应的测试用例如表 3-9 所示。

**表 3-9 NextDate 的有效等价类测试用例**

| ID | 输入数据(年-月-日) | 预期输出(年-月-日) | 备 注 |
|---|---|---|---|
| NextDate-EP-01 | 1979-2-14 | 1979-2-15 | 普通日期 Dt_1 |
| NextDate-EP-02 | 1981-12-31 | 1982-1-1 | 年末日期 Dt_21 |
| NextDate-EP-03 | 2008-6-30 | 2008-7-1 | 月末(30 天)Dt_221 |
| NextDate-EP-04 | 2010-10-31 | 2010-11-1 | 月末(31 天)Dt_222 |
| NextDate-EP-05 | 2016-2-29 | 2016-3-1 | 闰年 2 月末 Dt_2231 |
| NextDate-EP-06 | 2019-2-28 | 2019-3-1 | 非闰年 2 月末 Dt_2232 |

**5. 方法总结**

等价类划分法是一种有效地测试方法,能够将庞大的输入域化简为小规模的有限集合,且理论上讲化简前后的两个集合对测试而言是等效的。在有明确的条件和限制的情况下,利用等价类划分法可设计出完备的测试用例集,减少不必要的测试用例。

等价类划分法的优点是考虑了输入域的各类情况,避免了因盲目或随机选取测试数据而造成的覆盖不完整性或不稳定性。尽管等价类划分法比随机选取测试用例要优越,但仍存在不足。比如,等价类划分法往往只考虑独立条件的输入输出,而忽略多

个条件之间的关联关系；或者说，如果考虑了多个条件之间的关联关系，往往会产生很多的测试用例，造成非常高的测试代价。另外，等价类划分法还可能忽略掉特定类型的高效测试用例，比如某些边界值。边界值分析法和因果图法可弥补等价类划分法在这些方面的不足。

## 3.2.3 边界值分析

### 1. 概念定义

边界值分析法是指在被测对象的边界及边界附近设计测试用例的方法。测试实践发现，大量缺陷发生在被测对象的输入域或输出域的边界上，而不是在内部；如果针对边界附近加以重点验证，常常得到良好的测试效果。因此，边界值分析法不是选择等价类的任意元素，而是主要针对各种边界情况设计测试用例。

### 2. 原理分析

边界值分析法是对从等价类划分法导出的测试用例的一个非常合理的补充。程序错误经常出现在等价类的边界上，而出现这种情况的原因经常是没有明确定义边界值，或者编程人员对边界产生误解。利用边界值分析进行的测试往往能够有效地发现错误。

边界值分析法符合人们的一个基本假设：如果软件在能力达到极限的时候能够运行，那么它在正常情况下就不会有什么问题。

### 3. 测试设计

对于某个输入条件，边界值分析可参考以下原则：

(1) 若输入条件规定了取值范围，则以该范围为边界；

(2) 若输入条件规定了值的个数，则以值的个数为边界；

(3) 若输入域是有序集合(比如有序表、顺序文件等)，则选取集合中特定次序的数据作为边界，如第一个或最后一个数据等。

实际上，针对某个输入条件确定边界点时，可基于如下的思路。

(1) 在需求描述中寻找最大极限边界，比如，最高/最低、最多/最少、最前/最后等。对于凡是可用数值来描述的输入条件而言，无论在需求规格说明中是否明确指出其边界，其极限边界条件是一定存在的，因为在程序实现时，该输入条件一定需要用某个数据变量来表示，而数据变量对应的某个数据类型是有取值范围的。当需求规格说明中未明确规定输入条件的边界时，该取值范围就是其极限边界。

(2) 在需求描述中寻找其他较为明显的边界，这些边界的主要特征是：当在该点附近一个极小的邻域内分别取大于、等于和小于该点的3个值时，被测程序对它们的处理是不完全相同的。换言之，某个输入条件的边界不仅包含最小值点和最大值点，还可能存在其他非极值性质的边界点。

(3) 需要关注在软件内部的边界点，称为次边界条件或内部边界条件。比如，2的乘方、ASCII字符表等。软件最终用户一般是看不到这类边界的，寻找这类边界通常要求测试人员具有一定的编程经验。

**4. 实例演示**

以 NextDate 为例，分别考虑年、月、日的边界以及允许输入值的整体边界，设计边界值测试用例如表 3-10 所示。

**表 3-10　NextDate 的边界值测试用例**

| ID | 输入数据(年-月-日) | 预期输出(年-月-日) | 备注 |
| --- | --- | --- | --- |
| NextDate-BVT-01 | 1899-2-14 | 提示"请填写一个在 1900 和 2099 之间的整数" | 年的边界 |
| NextDate-BVT-02 | 1900-2-14 | 1900-2-15 | |
| NextDate-BVT-03 | 1901-2-14 | 1901-2-15 | |
| NextDate-BVT-04 | 2098-2-14 | 2098-2-15 | |
| NextDate-BVT-05 | 2099-2-14 | 2099-2-15 | |
| NextDate-BVT-06 | 2100-2-14 | 提示"请填写一个在 1900 和 2099 之间的整数" | |
| NextDate-BVT-07 | 2015-0-14 | 提示"请填写一个在 1 和 12 之间的整数" | 月的边界 |
| NextDate-BVT-08 | 2015-1-14 | 2015-1-15 | |
| NextDate-BVT-09 | 2015-2-14 | 2015-2-15 | |
| NextDate-BVT-10 | 2015-11-14 | 2015-11-15 | |
| NextDate-BVT-11 | 2015-12-14 | 2015-12-15 | |
| NextDate-BVT-12 | 2015-13-14 | 提示"请填写一个在 1 和 12 之间的整数" | |
| NextDate-BVT-13 | 2015-4-0 | 提示"请填写一个在 1 和 31 之间的整数" | 日的边界 |
| NextDate-BVT-14 | 2015-4-1 | 2015-4-2 | |
| NextDate-BVT-15 | 2015-4-2 | 2015-4-3 | |
| NextDate-BVT-16 | 2015-4-30 | 2015-5-1 | |
| NextDate-BVT-17 | 2015-4-31 | 提示"日期不存在" | |
| NextDate-BVT-18 | 2015-4-32 | 提示"请填写一个在 1 和 31 之间的整数" | |
| NextDate-BVT-19 | 1899-12-31 | 提示"请填写一个在 1900 和 2099 之间的整数" | 整体边界 |
| NextDate-BVT-20 | 1900-1-1 | 1900-1-2 | |
| NextDate-BVT-21 | 1900-1-2 | 1900-1-3 | |
| NextDate-BVT-22 | 2099-12-30 | 2099-12-31 | |
| NextDate-BVT-23 | 2099-12-31 | 2100-1-1 | |
| NextDate-BVT-24 | 2100-1-1 | 提示"请填写一个在 1900 和 2099 之间的整数" | |

**5. 方法总结**

边界值分析法是最基本、最简单的黑盒测试方法之一，通常作为等价类划分法的补充。

边界值分析法并不关心某个输入条件的边界点是被测对象可以接受的正常值，还是不能接受的错误值，而只关心该值是否是系统处理的边界。因此，在分析边界点和测试数据时，不需要特别考虑某个取值的有效性，只需在设计测试用例时针对有效和无效输入数据，分别设计其预期输出结果即可。

边界值分析法不仅可从问题的输入域寻找边界，还可以从问题的输出域寻找边界，特别是在输入域与输出域完全不相似的条件下，针对输出域的边界值测试尤其重要。从输出域的边界设计测试用例与从输入域的边界设计测试用例的原则一致，但仍存在一些不同之处，主要包括：如何选择合适的输出域来寻找边界点；如何限定边界点附近邻域的大小；针对边界值附近邻域内选中的测试数据，能够顺利确定对应的测试用例等。

## 3.2.4 因果图与决策表法

**1. 概念定义**

因果图和决策表法是一种黑盒测试方法，从分析软件需求规格说明开始，首先通过因果图得到因果列表，然后基于因果列表建立决策表，最后基于决策表生成测试用例。

因果图是一种形式语言，相当于一种数字电路，一个组合的逻辑网络，但没有使用标准的电子学符号，而是使用了简化的符号。因果图法是一种可以辅助测试人员明确测试对象，确定测试依据的有力手段，很好地弥补了等价类划分和边界值分析中未对输入条件的组合进行分析的缺点。因果图的基本符号如图 3-13 所示，C 表示原因，E 表示结果，各连接点表示状态，取值“0”表示状态不出现，取值“1”表示状态出现。

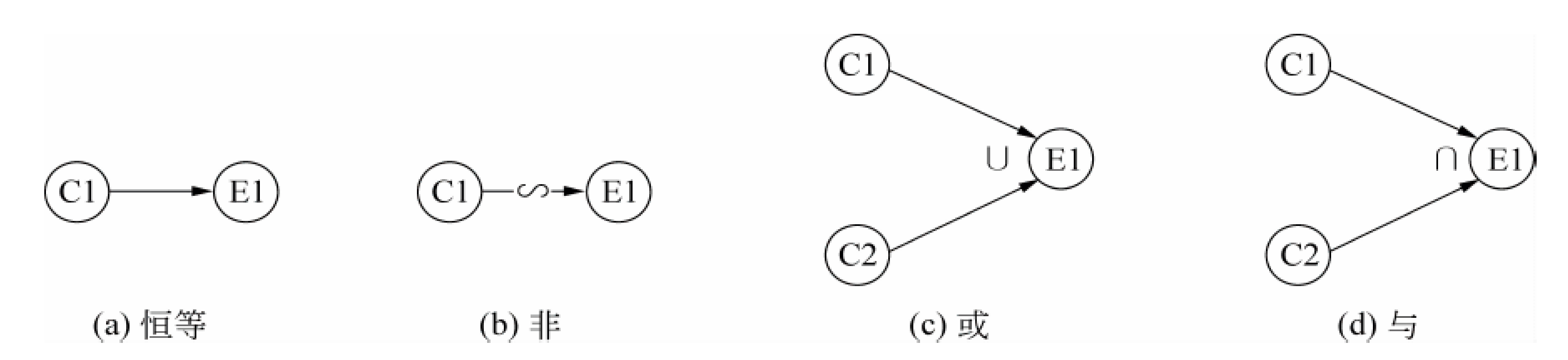



**图 3-13 因果图的基本符号**

因果图中原因与结果之间的关系包括如下 4 种。

(1) 恒等。表示原因与结果之间一对一的关系，若原因出现，则结果出现；若原因不出现，则结果不出现。

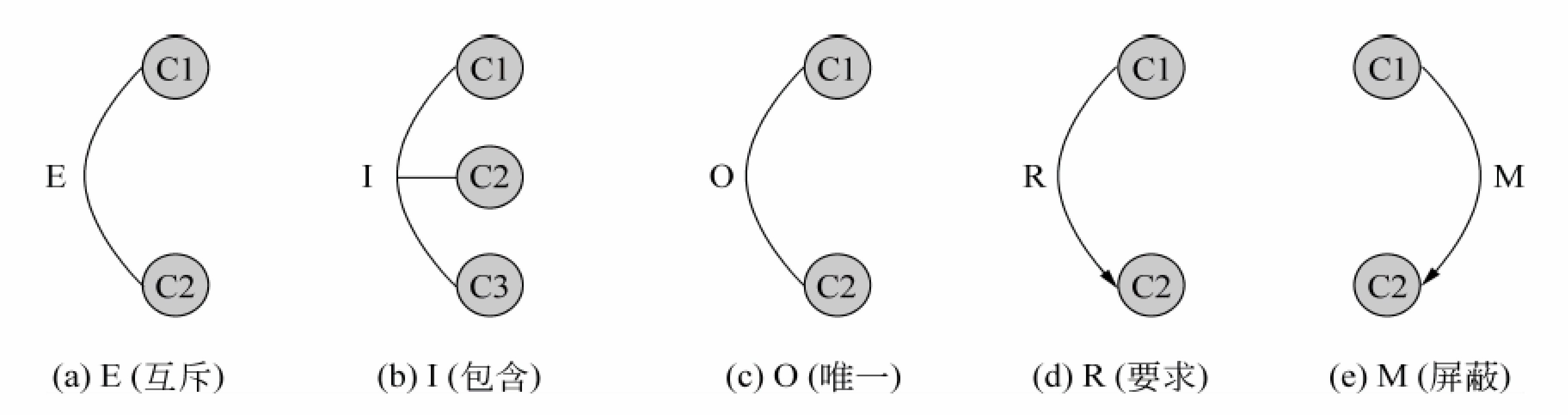
(2) 非。表示原因与结果之间的一种否定关系，若原因出现，则结果不出现；若原因不出现，则结果出现。

(3) 或。表示若几个原因中有一个出现，则结果出现；否则，结果不出现。

(4) 与。表示若几个原因都出现，则结果出现；否则，结果不出现。

另外，为了表示原因与原因之间的约束条件，定义约束符号如图 3-14 所示。共有 5 种约束，其中，“E(互斥)”表示两个中最多有一个成立；“I(包含)”表示至少有一个必须成立；

“O(唯一)”表示有且仅有一个成立;“R(要求)”表示当 C1 出现时 C2 必然出现,不可能 C1 出现且 C2 不出现;“M(屏蔽)”表示当 C1 为“1”时,C2 必须为“0”,而当 C1 为“0”时,C2 的取值不定。




(a) E (互斥)　(b) I (包含)　(c) O (唯一)　(d) R (要求)　(e) M (屏蔽)

**图 3-14　因果图的约束符号**

因果图必须转化成决策表,从决策表才可以直接得到测试用例。决策表由两部分组成,上半部分是输入区,下半部分是输出区,表中每一行是一个输入条件或预期输出的取值,每一列构成一个测试用例(各条件的组合以及针对这一输入组合的期望输出结果)。典型的决策表结构示例如表 3-11 所示。

**表 3-11　决策表的结构**

| 输入条件 A | A1 | A2 | A1 | A2 | A1 | A2 | A1 | A2 | A1 | A2 | A1 | A2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 输入条件 B | B1 | B1 | B2 | B2 | B3 | B3 | B1 | B1 | B2 | B2 | B3 | B3 |
| 输入条件 C | C1 | C1 | C1 | C1 | C1 | C1 | C2 | C2 | C2 | C2 | C2 | C2 |
| 预期输出 1 | × | | | × | × | | × | | | × | × | |
| 预期输出 2 | | × | × | | | × | | × | × | | | × |

**2. 原理分析**

在等价类划分和边界值分析方法中,对输入条件的考虑一般假设各条件之间是独立的,并未重视各输入条件的关联,即输入条件的可能组合。事实上,当输入条件存在可能的组合时,对这些组合加以考虑并检验在组合情况下程序是否能实现预期的功能是非常重要和必要的。要检查输入条件的组合不是一件容易的事情,即使把所有输入条件划分成等价类,它们之间的组合情况也非常多,因此需要一种适合于描述多种条件组合及其期望结果的测试用例设计方法。

因果图方法是一种根据条件的组合而生成测试用例的方法,它将规格说明转换为一个布尔逻辑网络,可使得测试人员从不同的视角更细致、更深入地来审视规格说明。事实上,建立因果图是一种暴露规格说明中模糊和不完整之处的好办法。

决策表由因果图产生,决策表的每一列都清楚地表明各输入条件及其取值的依赖关系,以及由这些输入组合得到的相应预期输出结果。决策表定义了逻辑测试用例,为了执行这些测试用例,必须输入具体的数据值并且标识前置条件和后置条件。

可以定义基于决策表的测试完成准则。最基本的要求是至少用一个测试用例来覆盖决策表中的每一列,这样就能验证所有关心的输入条件组合及其相应的输出结果。不过需要指出的是,在把每个组合设计成测试用例时,由于条件可能会相互影响甚至相互排斥,因此

不是所有的组合都是有效的。

决策表法的实质是对个体输入域的有效等价类测试的扩充，体现在如下方面：

（1）在输入区列出的所有取值并非具体的数据，而是每个输入条件的有效等价类，实际上是全部输入条件的强组合方式的等价类测试，因此只要有效等价类的划分不存在漏洞，则决策表可以保证有效输入域上完备的测试；

（2）决策表对输入条件仅考虑有效等价类，不考虑无效等价类，实际上针对无效等价类的测试一般不存在冗余，不需要考虑冗余的消除；

（3）在决策表的输出区，并不涉及输出域的等价类划分。

**3. 测试设计**

使用因果图和决策表法生成测试用例有如下 6 个基本步骤。

（1）分析规格说明，找出哪些是原因，哪些是结果，并对每个原因和结果进行唯一标识。所谓原因，是指进行等价类划分后的输入条件；所谓结果，是指输出条件。

（2）分析规格说明，找出原因和结果之间、原因和原因之间的对应或约束关系。

（3）绘制因果图。

（4）由于语法或环境限制，有些原因和原因之间、原因和结果之间的组合情况不可能出现。为表明这些特殊情况，在因果图上标记约束或限制条件。

（5）把因果图转换成决策表。又可分为如下步骤：①遍历每个结果进行逐一分析；②在因果图中查找能够得到这个结果的原因组合，以及不能产生这个结果的原因组合；③在决策表中，为每一个原因组合以及引起这个结果的状态加一列；④检查决策表条目是否出现冗余，如果有，则删除冗余的条目。

（6）依据判定表中的每一列设计测试用例。

**4. 实例演示**

对于 NextDate 程序，分别考虑年、月、日三个输入的等价类划分，获得原因共 11 个，以及结果 6 个。原因分别是：

M1＝{月|有 30 天的月份，即 4、6、9、11}；

M2＝{月|有 31 天的月份，但 12 月除外，即 1、3、5、7、8、10}；

M3＝{月|12 月}；

M4＝{月|2 月}；

D1＝{日|1≤日≤27}；

D2＝{日|28}；

D3＝{日|29}；

D4＝{日|30}；

D5＝{日|31}；

Y1＝{年|闰年，且 1900≤年≤2099}；

Y2＝{年|非闰年，且 1900≤年≤2099}。

结果分别是：

E1＝{日期加 1}；

E2＝{日期为 1(重置)}；

E3={月份加 1}；

E4={月份为 1(重置)}；

E5={年份加 1}；

E6={不可能情况}。

基于上面分析，可绘制因果图并基于因果图，得到 NextDate 的决策表如表 3-12 所示。

**表 3-12 NextDate 的决策表**

| 测试用例 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C1：月份在哪？ | M1 | M1 | M1 | M1 | M1 | M2 | M2 | M2 | M2 | M2 | M3 |
| C2：日期在哪？ | D1 | D2 | D3 | D4 | D5 | D1 | D2 | D3 | D4 | D5 | D1 |
| C3：年份在哪？ | — | — | — | — | — | — | — | — | — | — | — |
| E1：日期加 1 | √ | √ | √ |  |  | √ | √ | √ | √ |  | √ |
| E2：日期为 1 |  |  |  | √ |  |  |  |  |  | √ |  |
| E3：月份加 1 |  |  |  | √ |  |  |  |  |  | √ |  |
| E4：月份为 1 |  |  |  |  |  |  |  |  |  |  |  |
| E5：年份加 1 |  |  |  |  |  |  |  |  |  |  |  |
| E6：不可能情况 |  |  |  |  | √ |  |  |  |  |  |  |
| 测试用例 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| C1：月份在哪？ | M3 | M3 | M3 | M3 | M4 | M4 | M4 | M4 | M4 | M4 | M4 |
| C2：日期在哪？ | D2 | D3 | D4 | D5 | D1 | D2 | D2 | D3 | D3 | D4 | D5 |
| C3：年份在哪？ | — | — | — | — | — | Y1 | Y2 | Y1 | Y2 | — | — |
| E1：日期加 1 | √ | √ | √ |  | √ | √ |  |  |  |  |  |
| E2：日期为 1 |  |  |  | √ |  |  | √ | √ |  |  |  |
| E3：月份加 1 |  |  |  |  |  |  | √ | √ |  |  |  |
| E4：月份为 1 |  |  |  | √ |  |  |  |  |  |  |  |
| E5：年份加 1 |  |  |  | √ |  |  |  |  |  |  |  |
| E6：不可能情况 |  |  |  |  |  |  |  |  | √ | √ | √ |

在表 3-12 中，共有 22 条规则，仍可以进行简化，比如规则 1、2、3 分别涉及 M1 情况下的日期 D1、D2、D3，输出结果一致，是可以合并的。经过化简后的决策表如表 3-13 所示。相应的测试用例如表 3-14 所示。

**表 3-13 NextDate 的简化决策表**

| 测试用例 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C1：月份在哪？ | M1 | M1 | M1 | M2 | M2 | M3 | M3 | M4 | M4 | M4 | M4 | M4 | M4 |
| C2：日期在哪？ | D1～D3 | D4 | D5 | D1～D4 | D5 | D1～D4 | D5 | D1 | D2 | D2 | D3 | D3 | D4～D5 |

续表

| 测试用例 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C3：年份在哪？ | — | — | — | — | — | — | — | — | Y1 | Y2 | Y1 | Y2 | — |
| E1：日期加 1 | √ | | | √ | | √ | | √ | √ | | | | |
| E2：日期为 1 | | √ | | | √ | | √ | | | √ | √ | | |
| E3：月份加 1 | | √ | | | √ | | | | | √ | √ | | |
| E4：月份为 1 | | | | | | | √ | | | | | | |
| E5：年份加 1 | | | | | | | √ | | | | | | |
| E6：不可能情况 | | | √ | | | | | | | | | √ | √ |

**表 3-14　NextDate 的测试用例表**

| ID | 输入日期 | 预期输出 | 备注 |
|---|---|---|---|
| NextDate-DT-01 | 2015-6-27 | 2015-6-28 | 30 天月份的普通日期 |
| NextDate-DT-02 | 2015-6-30 | 2015-7-1 | 30 天月份的月末 |
| NextDate-DT-03 | 2015-6-31 | 提示日期不存在 | 无效日期 |
| NextDate-DT-04 | 2015-7-27 | 2015-7-28 | 31 天月份的普通日期 |
| NextDate-DT-05 | 2015-7-31 | 2015-8-1 | 31 天月份的月末 |
| NextDate-DT-06 | 2015-12-27 | 2015-12-28 | 12 月的普通日期 |
| NextDate-DT-07 | 2015-12-31 | 2016-1-1 | 12 月的月末(年末) |
| NextDate-DT-08 | 2015-2-27 | 2015-2-28 | 非闰年 2 月的普通日期 |
| NextDate-DT-09 | 2016-2-28 | 2016-2-29 | 闰年 2 月的普通日期 |
| NextDate-DT-10 | 2015-2-28 | 2015-3-1 | 非闰年 2 月的月末 |
| NextDate-DT-11 | 2016-2-29 | 2016-3-1 | 闰年 2 月的月末 |
| NextDate-DT-12 | 2015-2-29 | 提示日期不存在 | 无效日期 |
| NextDate-DT-13 | 2015-2-30 | 提示日期不存在 | 无效日期 |

### 5. 方法总结

因果图法有助于用一种较系统的方法选择出高效的测试用例集合，产生一组有效地测试用例，但通常不能生成全部所需的测试用例，因为它仍会受到各输入条件的等价划分的限制。另外，在条件的数量和依赖关系增加时，因果图和决策表的规模将增加的非常快，从而失去可读性。

当输入条件不存在相关性时，不需要使用因果图和决策表方法，因为等价类划分方法生成的测试用例之间本身就不存在冗余。另外，决策表法仅针对输入域展开分析，需要从设计的角度对输出域进行细化，才能得到更加准确的测试用例。

## 3.2.5 组合测试

**1. 概念定义**

待测系统运行时出现故障的原因可能是由于某一个参数的单独作用，但更多情况下是因为多个参数的相互作用。一般地，软件系统的输入数据或操作、内外部事件、软硬件配置等因素都可能触发软件故障，而由于影响因素多且每个因素的取值情况可能比较复杂，它们的相互组合情况将形成一个数量巨大的测试用例空间，给软件测试带来很大难度。

组合测试是一种充分考虑各种因素及其相互作用的软件测试方法，能够设计出一组数量较少但能满足参数组合覆盖要求的测试用例，是一种实用的技术方法。

**2. 原理分析**

设一个待测系统有 $n$ 个参数 $c_1 \in D_1, c_2 \in D_2, \cdots, c_n \in D_n$，$D_i(i=1,2,\cdots,n)$ 为参数 $c_i$ 的合理取值范围。遍历所有这些值是不可能也是不必要的，需要从中选出具有典型意义的离散点。一般地，先根据边界分析法选择 $c_i$ 可以取到的边界值，然后再根据等价类划分法补充若干取值点，使得 $c_i$ 的取值更具有代表性。经过这样处理后，参数 $c_1, c_2, \cdots, c_n$ 都取一组离散值 $T_1, T_2, \cdots, T_n$，其中 $T_i$ 表示 $c_i$ 可取的有限离散点集，并假设 $\alpha_i$ 表示 $c_i$ 可取值的个数，即 $\alpha_i = |T_i|$。

如果对这 $n$ 个参数的各种组合进行完全覆盖测试，需要 $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n$ 个测试用例，这在一般情况下是不可行的，所以转向对这些参数的部分组合进行覆盖测试。一般地，根据覆盖程度的不同，可以考虑单因素覆盖、两两组合覆盖、三三组合覆盖等。单因素覆盖指得是对每个参数的各个取值进行覆盖，要求每个参数的各个取值都至少出现一次，至少需要 $m = \max_{1 \leqslant i \leqslant n} \alpha_i$ 个测试用例；两两组合覆盖要求对任意两个参数的所有取值组合进行覆盖，保证每一个取值组合都至少出现一次，至少需要 $m = \max_{1 \leqslant i \neq j \leqslant n} \alpha_i \times \alpha_j$ 个测试用例；三三组合覆盖要求对任意三个参数的所有取值组合进行覆盖，至少需要 $m = \max_{1 \leqslant i \neq j \neq k \leqslant n} \alpha_i \times \alpha_j \times \alpha_k$ 个测试用例。依此类推，随着参数组合覆盖要求的增加，测试用例的数量也呈指数上升，测试代价越来越大；另一方面，如果某组测试用例能实现对任意 $l(1 \leqslant l \leqslant n)$ 个参数的组合覆盖，则也能实现对任意 $l-1$，$l-2, \cdots, 2, 1$ 个参数的组合覆盖，缺陷发现能力越来越强。在实际应用中，两两组合覆盖最为常用。

20 世纪初期工农业生产中经常用到正交试验设计方法，后来人们把正交试验法应用到软件测试中，利用正交表设计测试用例。随着研究的深入，人们发现在软件测试中各种组合的覆盖率不需要相等，完全可以利用覆盖表来替代正交表，随后 $k$ 维覆盖表在软件测试中得到很好应用。考虑到在实际软件系统中并不是任意几个参数间都存在相互作用，有些参数根本不会有任何交互，人们又提出可变力度覆盖表。正交表、$k$ 维覆盖表和可变力度覆盖表都是组合测试中常见的测试用例集，其关系如图 3-15 所示。

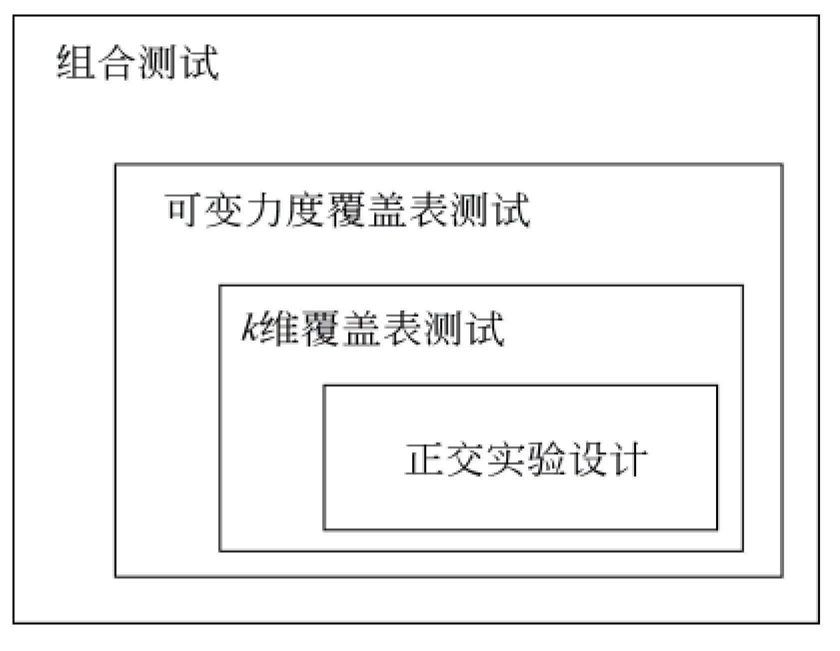



**图 3-15 组合测试与正交实验设计**

**3. 测试设计**

给出一种满足两两组合覆盖的测试生成方法。

假设待测系统共有 $m$ 个参数 $c_1, c_2, \cdots, c_m$，它们分别取离散值 $T_1, T_2, \cdots, T_m$，并设 $\alpha_j = |T_j|$ 为 $T_j$ 所含元素的个数，不妨设 $\alpha_m \leqslant \alpha_{m-1} \leqslant \cdots \leqslant \alpha_1 = \max_{1\leqslant j\leqslant m} \alpha_j \equiv n$。

1）得到矩阵 $\boldsymbol{C} = (c_{ij})_{n\times m}$

矩阵 $\boldsymbol{C}$ 由所有参数的取值组成。具体方法是，从 $c_1$ 开始，把它的 $n$ 个参数依次填入到 $\boldsymbol{C}$ 的第一列中，然后依次类推，把 $c_j$ 的 $\alpha_j$ 个参数依次填入到 $C$ 的第 $j$ 列中。若 $C$ 的某些参数所在的列有空位，则用该参数的任意取值填充之（对应于正常位，称这些空位为补充位，并称在补充位上填充的值为补充值），使得矩阵 $\boldsymbol{C}$ 的每列均有 $n$ 个元素。

2）展开矩阵

判定 $n$ 与 $m$ 的大小，如果 $n \geqslant m$，执行步骤（1）；否则，执行步骤（2）。

（1）$n \geqslant m$ 时，仿照矩阵分解的方法将矩阵 $\boldsymbol{C}$ 展开，分为如下两步：

① 先将矩阵 $\boldsymbol{C}$ 展开至最小矩阵。这里最小矩阵指的是 $(n-m)\times 1$ 矩阵（如果 $n>m$）或 $2\times 2$ 矩阵（如果 $n=m$）。

具体方法是，从第一列开始，用该列的每一个元素乘以剩余矩阵（划去该元素所在的行与列即得剩余矩阵），然后相加（称每一个以加号隔开的部分为一个测试项，简称项，下同）；并循环递推，直至展开到最小矩阵。在此过程中，需注意：如果在某测试项中最小矩阵外的地方出现了补充值，则应将该测试项剔除。

② 展开最小矩阵。将上一步得到的各项继续展开，得到最终的测试项，并记其集合为 $I$。在此过程中，需注意：

（a）$n>m$ 时，如果最小矩阵中的所有位都是补充位，则任选且仅选其中一个补充值与该项的前面部分组成最终项，添加到 $I$；否则，即最小矩阵中至少有一位正常位，则应分别把每一个正常位对应的值与该项的前面部分组成最终项，并把它们都添加到 $I$。

（b）$n=m$ 时，把最小矩阵中的补充位视为正常位，将矩阵展开后与该项的前面部分分别组成最终项，添加到 $I$。

（2）$n<m$ 时，先把矩阵 $\boldsymbol{C}$ 转置得到 $\boldsymbol{C}'$（矩阵 $\boldsymbol{C}'$ 的第 $i$ 行是 $\boldsymbol{C}$ 的第 $i$ 列），然后仿照矩阵分解的方法将 $\boldsymbol{C}'$ 展开，分为如下两步：

① 先将 $\boldsymbol{C}'$ 展开至最小矩阵，这里最小矩阵指的是 $(m-n)\times 1$ 矩阵。

② 展开最小矩阵。这一步与 $n \geqslant m$ 时不同。由于现在的最小矩阵的各个位置上为不同参数的值，我们只需把它们依次取出并与该项的前面部分组成最终项即可。

3）调整测试用例

（1）将矩阵 $\boldsymbol{C}$ 的每一行作为一项，添加到 $I$ 中。注意：如果某行除第一个位置外都是补充位，那么此行不应作为一项。

（2）由于补充位的原因，$I$ 中可能存在重复项。检查 $I$，如果有重复项，将其剔除。

至此，我们得到的 $I$ 即为最终的测试用例的集合，$I$ 中的每一项就是一个测试用例。

**4. 实例演示**

设待测系统的参数为 $x$，$y$，$z$，参数取值分别为 $x$：$a_1$，$a_2$，$a_3$，$a_4$；$y$：$b_1$，$b_2$，$b_3$；$z$：$c_1$，$c_2$。则按照算法，可以得到矩阵 $\boldsymbol{C}$ 并将其展开如下（右上角加 * 表示补充位）：

$$
\boldsymbol{C}=\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_1^* \\ a_4 & b_1^* & c_2^* \end{pmatrix}=a_1\begin{pmatrix} b_2 & c_2 \\ b_3 & c_1^* \\ b_1^* & c_2^* \end{pmatrix}+a_2\begin{pmatrix} b_1 & c_1 \\ b_3 & c_1^* \\ b_1^* & c_2^* \end{pmatrix}+a_3\begin{pmatrix} b_1 & c_1 \\ b_2 & c_2 \\ b_1^* & c_2^* \end{pmatrix}+a_4\begin{pmatrix} b_1 & c_1 \\ b_2 & c_2 \\ b_3 & c_1^* \end{pmatrix}
$$

$$
=a_1b_2\begin{pmatrix} c_1^* \\ c_2^* \end{pmatrix}+a_1b_3\begin{pmatrix} c_2 \\ c_2^* \end{pmatrix}+a_2b_1\begin{pmatrix} c_1^* \\ c_2^* \end{pmatrix}+a_2b_3\begin{pmatrix} c_1 \\ c_2^* \end{pmatrix}+a_3b_1\begin{pmatrix} c_2 \\ c_2^* \end{pmatrix}+a_3b_2\begin{pmatrix} c_1 \\ c_2^* \end{pmatrix}+a_4b_1\begin{pmatrix} c_2 \\ c_1^* \end{pmatrix}
$$

$$
+a_4b_2\begin{pmatrix} c_1 \\ c_1^* \end{pmatrix}+a_4b_3\begin{pmatrix} c_1 \\ c_2 \end{pmatrix}
$$

$$
=a_1b_2c_1+a_1b_3c_2+a_2b_1c_1+a_2b_3c_1+a_3b_1c_2+a_3b_2c_1+a_4b_1c_2+a_4b_2c_1+a_4b_3c_1+a_4b_3c_2
$$

从而，$I=\{a_1b_2c_1, a_1b_3c_2, a_2b_1c_1, a_2b_3c_1, a_3b_1c_2, a_3b_2c_1, a_4b_1c_2, a_4b_2c_1, a_4b_3c_1, a_4b_3c_2, a_1b_1c_1, a_2b_2c_2, a_3b_3c_1\}$，共 13 个测试用例。

**5. 方法总结**

组合测试考虑了待测软件的各种因素及其相互作用关系，能够设计出一组数量较少但能满足一定参数组合覆盖要求的测试用例。研究表明，不超过 30％的缺陷是由 3 个及 3 个以上参数的相互作用引起的，几乎不存在 6 个以上参数相互作用引发的故障，因此组合测试是能够近似等效于完全测试的。

组合测试可只基于需求规格说明，只需要提取可能影响软件的参数及其取值情况，不需要过多的实现细节，且易于自动化实现，是一种好用的黑盒测试方法。

组合测试也有一些不足，比如：组合测试仍是一种不完全测试，可能存在测试不充分的风险；如果待测系统的参数及其取值选择的不够恰当，组合测试的作用将大打折扣；如果没有完整的预期输出，组合测试的效果也难以体现。

## 3.2.6 基于场景测试

**1. 概念定义**

基于场景测试的测试由 IBM Rational 公司较早提出，借鉴了软件设计时用事件触发控制流程形成软件使用场景的思想。同一事件不同的触发顺序和处理结果就形成事件流，基于场景测试以事件流为核心，可以比较生动地描绘出事件触发时的场景，有利于测试设计人员设计测试用例，同时使得测试用例更易理解和执行。

基于场景测试是高层测试设计的基础，通常用于功能测试中。

**2. 原理分析**

基于场景测试以事件流为核心，通过分析不同事件的触发顺序和处理结果，模拟操作系统使其从初始状态转变到结束状态的过程，形成不同的测试场景。

事件流分为基本流和备选流。基本流是从系统的某个初始状态开始，经一系列状态变化后到达终止状态的过程中，最主要的一个业务流程。所谓基本，指的是该事件流是整个业务流程中最基本的一个流程。基本流所涉及的业务规则并一定很复杂，但通常被认为是一条最高风险的业务流程，比如，所包含的功能点是用户最常使用的，对应的用户操作是需求中规定的正常操作，能够反映绝大多数用户操作被测软件的顺序，或者，该流程所包含的功

能点涉及某个复杂的核心算法，必须确保其正确性等。

备选流是以基本流为基础，在经过基本流上每个判定节点处(包含条件判定和循环判定)，满足不同的触发条件而导致的其他事件流。与基本流是一条从初始状态到终止状态的完整业务流程不同，备选流仅仅是业务流程的一个执行片段。基于一个基本流将得到多个备选流，备选流的数目取决于基本流上判定节点的数目与事件分析的颗粒度。

**3. 测试设计**

基于场景测试的主要步骤包括以下 4 步。

(1) 根据需测试的业务，分析基本流和备选流。

(2) 基于事件流，构建场景。

(3) 根据场景，设计逻辑测试用例。每个场景对应于业务流程中从开始状态到结束状态的一系列操作或执行步骤，可转化为测试用例的输入条件和预期输出。对应每个场景可设计一个或多个测试用例，基本步骤包括：①根据场景的流程，分析系统应满足的所有输入条件和预期输出；②当场景中包含备选流时，应确定该备选流触发的输入条件并予以标记。

(4) 根据逻辑测试用例，设计物理测试用例。对于步骤(3)设计的逻辑测试用例，应使用等价类划分、边界值分析等方法，设计物理测试用例。

**4. 实例演示**

结合一个简化的银行自动柜员机(ATM 机)系统进行介绍。

1) 根据需测试的业务，分析基本流和备选流

基本流的初始状态是 ATM 机系统已就绪，屏幕显示欢迎使用界面。后续业务流程包括如下步骤。

(1) 插卡：用户将银行卡插入 ATM 机的银行卡槽中。

(2) 卡校验：ATM 机系统读取插入的卡片信息，判断其是否为可接收的卡。在基本流中，卡片通过校验，进入下一步。

(3) 输入密码：在 ATM 机系统要求后，用户输入银行卡密码。

(4) 密码校验：ATM 机系统将用户输入密码与卡片预设密码进行比对，判断密码是否正确。在基本流中，输入的密码正确，进入下一步。

(5) 选择取款交易：根据 ATM 机系统的提示，用户在界面选择取款交易。

(6) 设置取款金额：根据 ATM 机系统的提示，用户在界面输入取款金额。输入的取款金额具有一定的限制条件，比如应为 10 的整数倍、不能超出日限额等。

(7) 交易校验：ATM 机系统将账户信息、交易类型、交易金额等作为一笔交易发送银行系统进行校验，判断能否允许交易进行。在基本流中，系统处于联机状态，银行批准完成交易并更新账户余额。

(8) 吐出钞票：ATM 机系统从现金槽中提供用户要提取的钞票。

(9) 退卡：ATM 机系统返还用户的银行卡。

至此用例结束，ATM 机系统再次回到准备就绪状态。

根据基本流中的 3 个关键校验点，得到 4 条备选流如下。

备选流 1：卡错误。在基本流步骤(2)处触发，ATM 机系统进行卡校验时判断银行卡无效，将卡片退回并提示卡无效，系统回到准备就绪状态，本用例结束。

备选流 2：密码错误。在基本流步骤(4)处触发，ATM 机系统在进行密码校验时为用户提供 3 次密码输入机会，当前两次密码输入错误后，ATM 机系统提示密码错误，返回密码输入状态，在基本流步骤(3)处重新加入基本流。

备选流 3：密码校验失败。在基本流步骤(4)处触发，ATM 机系统在进行密码校验时为用户提供 3 次密码输入机会，当密码第 3 次输入错误后，系统提示密码失败，退卡(真实情况下将会吞掉银行卡并提示用户到银行柜台办理取卡事宜)，系统回到准备就绪状态，本用例结束。

备选流 4：取款金额错误。在基本流步骤(7)处触发，ATM 机系统检验用户输入的取款金额错误时，禁止取款，要求用户重新输入取款金额，系统返回金额输入状态，在基本流步骤(6)处重新加入基本流。

自动柜员机系统的基本流和备选流如图 3-16 所示。

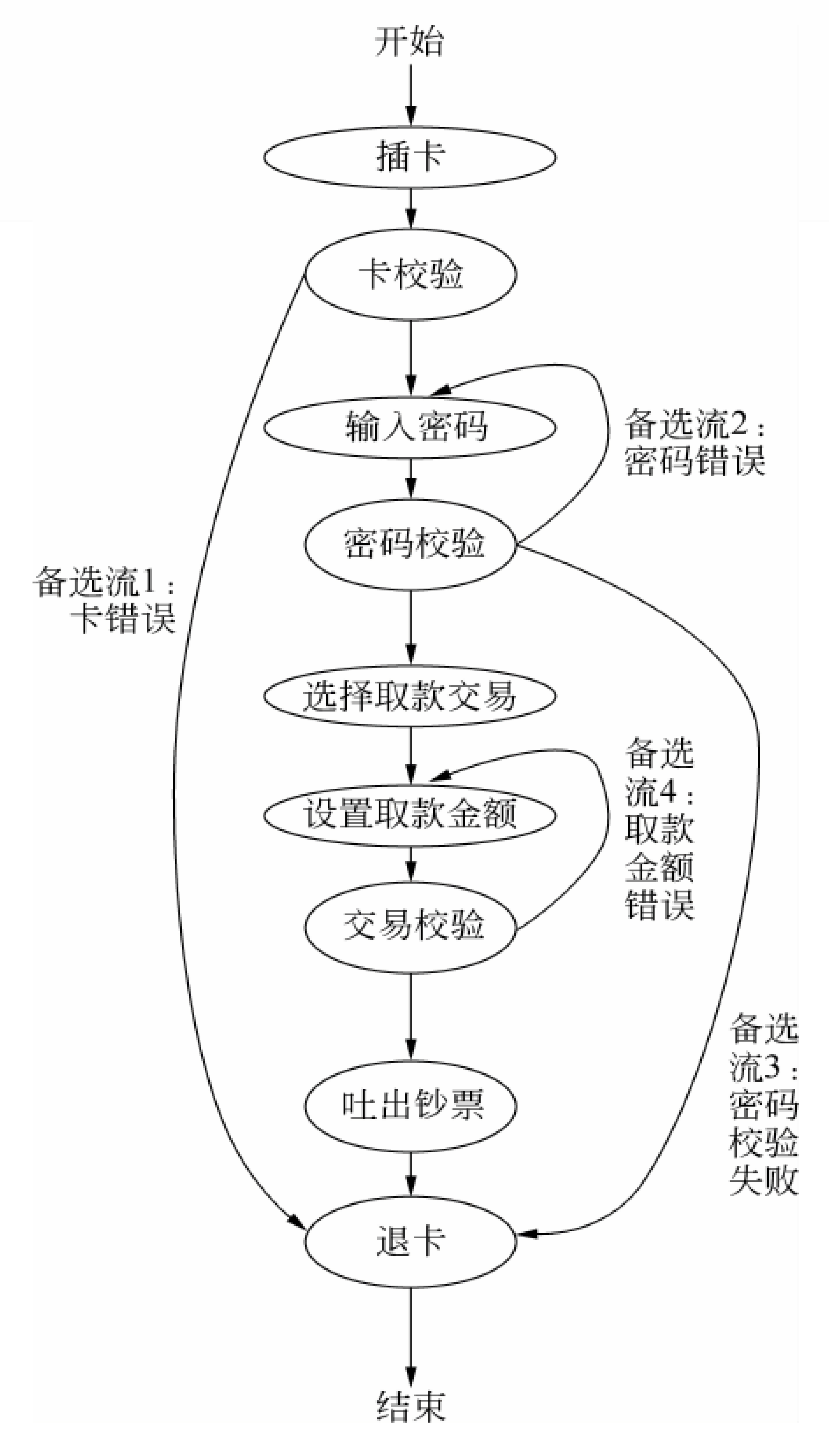



**图 3-16　自动柜员机系统的基本流和备选流**

2) 基于事件流，构建场景

根据步骤 1)中的基本流和备选流，构建 5 个场景，分别如下。

场景 1：取款成功，对应于基本流。

场景 2：卡错误，对应于基本流与备选流 1。

场景 3：密码错误，对应于基本流与备选流 2。

场景 4：密码校验失败，对应于基本流与备选流 3。

场景 5：取款金额错误，对应于基本流与备选流 4。

3）根据场景，设计逻辑测试用例

对于 ATM 机系统，设计测试用例如表 3-15 所示，表中的“V”表示该条件有效，“X”表示其是触发对应备选流的条件。

**表 3-15 自动柜员机场景测试的逻辑测试用例**

| 场景 | 输入 | | | | 预期输出 |
|---|---|---|---|---|---|
| | 账户 | 密码 | 取款金额 | 账户余额 | |
| 1 | V | V | V | V | 取款成功 |
| 2 | X | — | — | — | 消息提示，卡片退回，交易失败 |
| 3 | V | X | — | — | 消息提示，返回基本流步骤(3) |
| 4 | V | X | — | — | 消息提示，卡片退回，交易失败 |
| 5 | V | V | X | V | 消息提示，返回基本流步骤(6) |

检查设计的测试用例集，查看是否所有输入条件都取到“X”的情况，即至少一次作为测试用例的触发条件，否则测试用例集可能存在漏洞，应进行分析。在表 3-15 中，除账户余额外，其余输入条件都取到过“X”，而账户余额不属于用户界面输入项，不存在漏洞。

4）根据逻辑测试用例，设计物理测试用例

对于 ATM 机系统，假设存在一个有效账户 9999-2016-0521，对应密码是 1234。设计物理测试用例如表 3-16 所示。还可使用等价类划分、边界值分析等方法，设计更加完备的测试用例集。

**表 3-16 自动柜员机场景测试的物理测试用例**

| 场景 | 输入 | | | | | 预期输出 |
|---|---|---|---|---|---|---|
| | 账户 | 密码 | 取款金额 | 账户余额 | ATM余额 | |
| 1 | 9999-2016-0521 | 1234 | 100 | 1000 | 10000 | 取款成功 |
| 2 | 8888-2016-0521 | — | — | — | 10000 | 消息提示，卡片退回，交易失败 |
| 3 | 9999-2016-0521 | 4321 | — | — | 10000 | 消息提示，返回基本流步骤(3) |
| 4 | 9999-2016-0521 | 4321<br>4231<br>1243 | — | — | 10000 | 消息提示，卡片退回，交易失败 |
| 5 | 9999-2016-0521 | 1234 | 2000 | 1000 | 10000 | 消息提示，返回基本流步骤(6) |

### 5. 方法总结

与等价划分、边界值分析等方法主要侧重于测试数据的选择，关注输入取值而不涉及操作的动态执行过程不同，基于场景的测试以事件流为核心，通过分析不同事件的触发顺序和

处理结果，构建并基于事件流，形成不同的测试场景，最终基于测试场景设计测试用例。

由于场景测试从全局把握系统的整个业务流程，能够保证在多功能点交叉、存在复杂约束条件等情况下，实现测试的充分覆盖性，因此尤其适合于高层测试用例的设计。

基于场景测试的主要难点在于如何根据业务实际提取基本流，以及如何很好地控制备选流的数量。构建场景时，可能存在逻辑上可行而事实上不可能实现的场景。设计测试用例时，应注意一个测试用例唯一对应于一个场景，但一个场景可能需要设计多个测试用例。

基于场景测试仅需针对输入域进行分析，不适用于从输出域展开测试。

## 3.2.7 错误推测法

### 1. 概念定义

错误推测法是基于测试人员的经验和直觉来推测被测系统中可能存在的各种缺陷，进而有针对性地设计测试用例的方法。这里的经验和直觉来自测试人员对被测系统特性的了解及其在以往测试工作中的经历总结。

### 2. 原理分析

错误猜测法的基本思想是根据测试人员的经验和直觉，列举出被测系统中所有可能存在的缺陷和容易发生缺陷的情况，然后针对它们，直接设计和选择测试用例。

系统缺陷的列举和测试用例的设计往往不采用系统的和完备的方法，而是主要依赖于测试人员的经验和直觉。

### 3. 测试设计

由于错误猜测主要是依赖于直觉的非正规化的过程，很难描述出这种方法的规程。

错误猜测法的基本思想是列举出可能存在的缺陷及发生这些缺陷的常见情况的清单，然后依据清单来设计测试用例。一般地，可从以下几个方面进行错误猜测：软件产品以前版本中存在的问题，受开发语言、中间件、操作系统、浏览器、网络设施等环境的限制而可能带来的问题，对非法的、错误的和无意义的数据的识别、容错和处理而产生的问题，针对代码资源分配、内存泄漏等的设计等。

### 4. 实例演示

表 3-17 列出了对若干常用功能使用错误推测法的部分测试点，供读者参考。

表 3-17 部分常见功能的错误猜测法测试点

| 功能名称 | 功能简介 | 测试点 |
|---|---|---|
| 输入验证 | 主要包括数字输入验证、非法字符输入验证、输入长度验证、必填项验证与信息提示等 | 数字输入验证：分别输入数字(正数、负数、零值、单精度、双精度)、字符串、空白值、空值、临界数值、不合法的输入，系统给出必要的判断提示信息 |
| | | 时间、日期输入验证：分别输入任意字符、任意数字、非日期格式的数据、非正确日期(错误的闰年日期)、空值、空白值、不合法的输入，系统给出必要的判断提示信息 |
| | | 多列表选择框：测试是否能够多选、列表框中的数据是否能够显示完整。当列表框中的数据过多时，需要对数据有一定格式的排序 |

续表

| 功能名称 | 功 能 简 介 | 测 试 点 |
|---|---|---|
| 输入验证 | 主要包括数字输入验证、非法字符输入验证、输入长度验证、必填项验证与信息提示等 | 单列表下拉框：测试是否能够手工输入、下拉框中的数据是否能够显示完整。当下拉框的数据很多时，需要对数据有一定格式的排序。当下拉框的数据非常多时，下拉框可能会超出显示范围，这是不能被接受的 |
| | | 大文本输入框：它能够满足大数据量的输入，但最好能够显式地标明输入字符的长度限制，并且应该结合字符输入验证进行。需要注意的是，应该允许标点符号的存在 |
| | | 文件输入框：主要用于文件的上传操作，测试时应注意输入文件的扩展名，上传的文件大小应进行限制，不宜过大。从测试角度来说，开发人员必须对扩展名进行输入限制，并且在适当的地方对输入格式进行提示。当输入是空值等不合法的值时，系统给出必要的判断提示信息 |
| | | 输入字符长度验证：输入字符的长度是否超过实际系统接收字符长度的能力。当输入超出长度时，系统给出必要的判断提示信息 |
| | | 必填项验证：输入不允许为空的时候，系统需要有提示用户输入信息功能 |
| | | 格式、规则输入验证：当需要按照规定格式或指定规则进行输入时，系统需具有提示用户输入信息功能 |
| | | 系统错误定位的输入验证：当输入存在问题被系统捕获到时，页面的光标能够定位到发生错误的输入框 |
| | | 单选框、多选框的输入验证：单选框需要依次验证单选框的值是否都有效；多选框需要依次验证多选框的值是否都有效 |
| | | 验证码验证：应先结合字符输入验证进行测试注意当利用浏览器回退或者刷新时，显示的验证码应该和实际系统验证码一致。如果验证码以图片形式显示，但图片不能看到或显示不完整时，系统应允许进行重新获取，而不是整个页面刷新 |
| 操作验证 | 操作验证主要针对页面操作 | 页面链接检查：每一个链接是否都有对应的页面，并且页面之间切换正确 |
| | | 按钮功能检查：增加、删除、更改、查询、提交等按钮功能是否正确 |
| | | 重复提交表单：一条已经成功提交的记录，用浏览器回退后再提交，检查系统是否进行了处理 |
| | | 多次浏览器回退：检查多次使用浏览器回退的情况，在回退的地方再回退，重复多次，检查是否出错 |
| | | 快捷键检查：是否支持常用快捷键，比如 Ctrl+C、Ctrl+V 等。对一些不允许信息手动输入的字段，如日期选择等，是否对快捷键进行了限制 |
| | | 回车键检查：在输入结束后直接键入回车键，检查系统如何处理 |
| | | 上传下载文件检查：上传下载文件的功能是否实现，对上传文件的格式有何规定，下载文件是否能够打开，系统是否有解释信息 |
| | | 其他验证：在页面上的图片的大小是否合适，需要第三方软件支持的，应该给予必要的提示信息等 |

续表

| 功能名称 | 功能简介 | 测试点 |
|---|---|---|
| 登　录 | 主要针对登录功能，需结合访问控制验证进行 | 登录名输入：进行“输入验证”，需注意登录名是否区分大小写和空格 |
| | | 密码输入：进行“输入验证” |
| | | 提交操作：当输入正确的登录名和密码后，应能够进入到指定的页面；当输入的登录名或密码有误时，系统限制其登录，给出适当的提示信息且不应透漏关键信息。当遇到错误时，应进行错误页面测试 |
| | | 重设操作：当进行重设操作时，当前页面上所有的输入项被清空 |
| 新　增 | 主要针对页面数据增加操作 | 添加输入内容：进行“输入验证” |
| | | 重复增加验证：系统应该限制重复添加。利用网络传输以及服务器的延迟等，通过多次单击“增加”按钮等进行重复提交操作，经常会在数据库中发现重复数据 |
| | | 操作结果提示：当操作成功或失败时，应该给出必要的提示信息 |
| | | 文件数据的增加：有些增加包含了数据库数据的增加和一些文件的增加，此时的数据会保存在多个地方，需要对相关的数据做全面的验证 |
| | | 文件数据的验证：进行“输入验证”和“文件输入框输入验证”。当上传的文件为中文名称时，上传到服务器后可能会出现乱码现象 |
| 删　除 | 主要针对页面数据删除操作 | 选择需要删除的数据字段：一般要求系统按照 ID 来删除整条记录，因为按照名称来删除时，可能会遇到重名的问题 |
| | | 重复删除验证：系统应该限制重复删除。利用网络传输以及服务器的延迟等，通过多次单击“删除”按钮等进行重复删除操作，经常会造成数据库崩溃 |
| | | 关联数据删除：当被删除的数据还关联有文件时，需验证数据库中的数据以及硬盘下的文件是否都被同时删除 |
| | | 操作结果提示：当数据删除成功或失败时，应该给出必要的提示信息 |
| 修　改 | 主要针对页面数据修改操作 | 修改页面验证：打开需要修改的数据页面，注意与新增页面相比，只能修改部分数据，有些关键字是不应允许被修改的，并且两者数据应该是一致的 |
| | | 输入限制验证：修改页面上的输入限制应与新增页面上的输入限制一致 |
| | | 操作结果提示：当数据修改成功或失败时，应该给出必要的提示信息 |
| 查　询 | 主要针对页面查询操作 | 条件输入验证：条件输入查询，应先进行条件输入框的“输入验证” |
| | | 条件组合查询：将多个条件进行组合查询，结果可以通过数据库验证。需要注意的是，如果遇到某天的查询时间段，有的数据库认为一天不包括零点零分，有的数据库认为包括 |
| | | 查询结果显示：所有的查询结果应按照一定顺序排列，比如按照名称或 ID 等 |
| | | 操作结果提示：当数据查询成功或失败时，应该给出必要的提示信息 |

续表

| 功能名称 | 功能简介 | 测试点 |
|---|---|---|
| 翻页 | 主要针对页面翻页操作 | 分页显示验证：当数据量很大时，需进行分页显示，每页显示的行数应有限制，每行应有序号标识，行与行之间颜色要有一定的区分 |
| | | 翻页按钮：翻页按钮应该包括“首页”“上一页”“下一页”“尾页”以及“页面跳转”输入框等，按钮要能正常显示和使用 |
| | | 数据显示准确性验证：每页显示的数据要准确，确保没有查不出的数据，最好的做法是和数据库结合起来验证 |
| | | 页码数显示验证：页面太多以至于翻页数据不能全部显示时，应有完善的应对机制，比如只显示当前页的前 3 页和该页的后 3 页的页码数 |
| | | 页码标识验证：当翻到某页时，系统应该有明显的标识，标出该页面所处的页码 |
| 错误页面 | 主要针对系统异常情况下生成的错误页面 | 提示页面显示：当系统出现错误时，不应让服务器的调试信息出现在页面上，而应给出一个合适的提示信息 |
| | | 友好错误页面：当由于系统繁忙等原因，无法及时给出正确反馈信息时，系统应给出友好的提示页面，比如“请用户稍后再试”等提示信息 |

**5. 方法总结**

有些测试人员似乎天生就是测试的能手，不需要使用因果图、边界值等特殊的方法，就能够迅速发现被测系统中存在的问题，甚至是别人很难发现的问题，这其实就是错误猜测法的实践。

错误猜测法偏重于考虑系统的非正常流程、对异常数据的处理和其他一些异常情况，有时能够发现系统中不易被发现的缺陷，但是用好错误猜测法也是不容易的，一般需要测试人员能够充分地理解系统业务、具有丰富的开发和测试经验，甚至掌握全面的测试技术。

另外，由于错误猜测法不是一种完备的方法，容易造成测试内容或测试点的遗漏，所以经常作为其他方法的补充，与其他方法一起使用。

### 3.2.8 黑盒测试技术分析

黑盒测试方法主要通过分析规格说明中被测软件输入和输出的有关描述来设计测试用例，不需要了解被测软件的实现细节，因此方法简单，适用面广。总的来说，黑盒测试的优点和缺点都是明显的。

黑盒测试的优点主要在于：

(1) 对于大规模代码，黑盒测试比白盒测试更加高效；

(2) 黑盒测试一般不由开发人员进行，相对更客观；

(3) 黑盒测试有助于从软件设计开发的一开始就建立质量准则；

(4) 黑盒测试不依赖于程序内部结构，与软件开发相对分离，可以并向或提前开展；

(5) 不要求测试人员具备极高的技术能力，不要求掌握软件开发知识；

（6）有助于识别规格说明中含糊、矛盾或歧义的地方。

黑盒测试的缺点主要在于：

（1）在没有清晰的规格说明的情况下，设计测试用例是困难和具有挑战性的；

（2）测试用例设计一般不以软件设计说明等为基础，有时很难发现一些微妙的输入；

（3）在测试时间有限的情况下，很难识别和测试所有的输入；

（4）在黑盒测试中可能存在未被识别的路径；

（5）生成良好的测试用例是耗时和困难的工作；

（6）不能发现规格说明的错误；

（7）很多情况下，需要手工进行测试。

黑盒测试方法不仅包括本文介绍的方法，还包括错误推测法、正交试验法、功能图法、场景法以及元素分析法、语法测试等多种方法，限于篇幅，在此不详细介绍。在实际测试工作中，测试人员应充分了解用户需求和被测软件系统特性，灵活选用合适的测试方法，才能达到最佳的测试效果。最后，作为建议，给出黑盒测试方法的一般选择策略：

（1）进行等价类划分，包括输入条件和输出条件的等价划分，将无限测试变成有限测试，这是减少工作量和提高测试效率最有效的方法；

（2）在任何情况下，都必须使用边界值分析法，经验表明，用这种方法设计出的测试用例发现程序错误的能力最强；

（3）可以使用错误推测法追加一些测试用例，这需要依靠测试工程师的智慧和经验；

（4）对照程序逻辑，检查已设计出的测试用例的逻辑覆盖程度，如果没有达到要求的覆盖标准，应当再补充足够的测试用例；

（5）如果程序的功能说明中含有输入条件的组合情况，则一开始就可以选用因果图与决策表法；

（6）对于参数配置类的软件，要用组合测试或正交试验法选择较少的组合方式达到最佳效果；

（7）对于业务流清晰的系统，可以利用场景法贯穿整个测试设计过程，并在测试中综合使用各种测试方法。

# 3.3　灰盒测试

## 3.3.1　概述

灰盒测试是将白盒测试、黑盒测试结合起来的一种无缝的测试方法，使用基于规格说明的测试用例，对软件是否满足外部规格说明进行确认，并运行和验证软件所有路径，是一种全生存周期的测试方法。

白盒测试和黑盒测试各有其自身优点，也都存在难以克服的不足，主要表现在只考虑了软件程序某一方面的属性和特征，在前面已有分析。这样，要进行全面的程序测试，不得不把测试工作分两次进行，用白盒方法测试一次，再用黑盒方法测试一次，不但浪费时间，而且效果不一定好。灰盒测试正是基于此提出的，它较好地综合了白盒测试和黑盒测试的优点，

克服了部分白盒测试和黑盒测试的缺点。

灰盒测试是一种综合测试方法,基于程序运行时的外部表现并结合程序内部逻辑结构来设计测试用例,采集程序外部输出和外部接口数据以及路径执行信息来衡量测试结果,对软件程序的外部需求及内部路径都进行检验。

灰盒测试方法以软件程序的功能需求和性能指标为测试依据,主要根据需求规格说明、程序流程图以及测试人员经验来进行测试设计。相比于白盒测试而言,灰盒测试更接近黑盒测试。

灰盒测试是一种全生存周期的测试方法,可用于多阶段的测试。最常见的是用于集成测试中,重点关注软件系统的各个模块之间的相互关联,即模块之间的互相调用、数据传递、同步/互斥关系等。

## 3.3.2 实施步骤

灰盒测试前,需做好充分准备。除部署被测程序外,还需要具备源代码,准备好编译与运行环境,以及代码覆盖率工具。

实施灰盒测试,一般主要包括如下步骤(如图 3-17):

(1) 识别被测程序的输入输出;

(2) 识别被测程序的各种状态;

(3) 识别被测程序的执行路径;

(4) 识别被测程序的软件功能;

(5) 设计测试用例(包括输入和期望结果等)集合;

(6) 执行测试用例并检验执行结果。

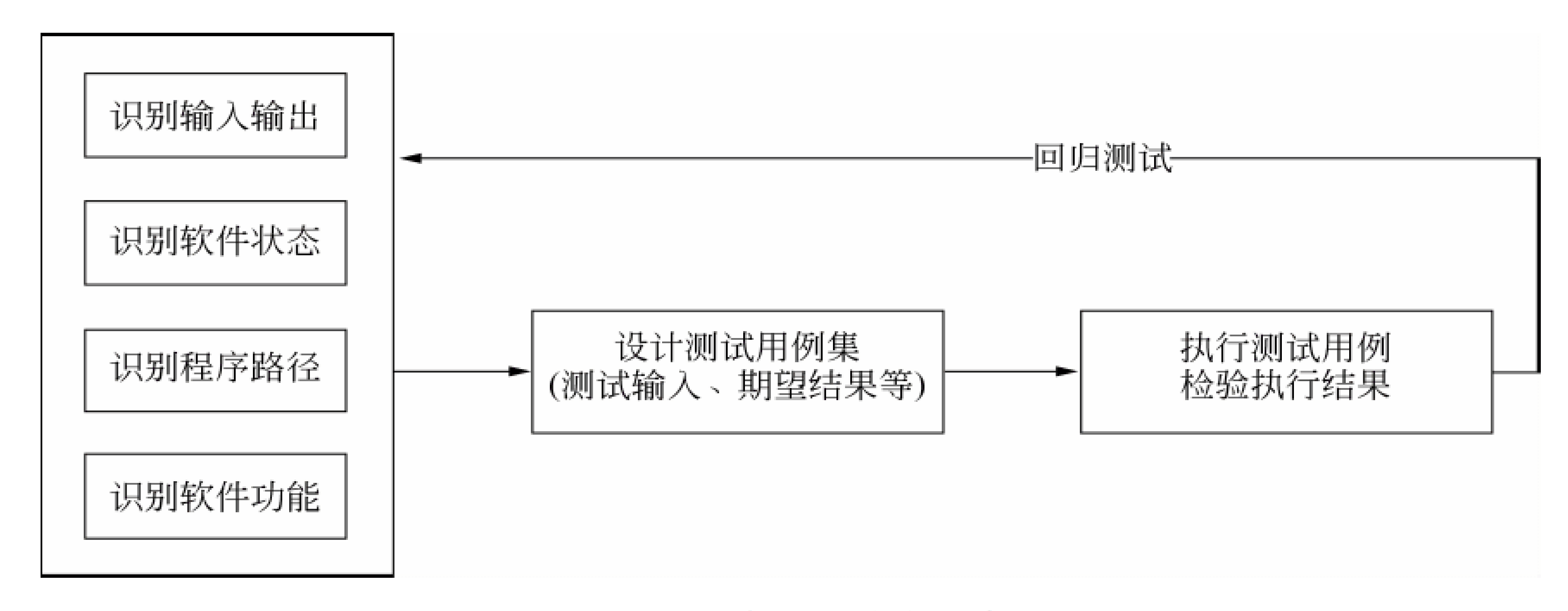



**图 3-17 灰盒测试的主要步骤**

## 3.3.3 灰盒测试技术分析

在软件测试领域,灰盒测试属于较新的测试技术。DO-178B 规范中新近加入了利用灰盒测试方法进行测试的标准。与白盒测试和黑盒测试相比,灰盒测试有以下特性:

(1) 灰盒测试根据规格说明来设计测试用例,这同黑盒测试一样,但要深入到系统内部

的特殊点同时进行功能测试和结构测试；

(2) 灰盒测试需要了解程序结构和代码实现，这同白盒测试一样，但一般无须关心函数或程序单元内部的实现细节；

(3) 在执行时，灰盒测试一般地需要通过编写代码，调用函数或者封装好的接口的方式进行；

(4) 灰盒测试通常在白盒测试之后、进行大规模集成测试之前，由测试人员执行。

总的来说，灰盒测试的优点主要有：

(1) 能够进行基于需求的覆盖测试和基于程序路径的覆盖测试；

(2) 测试结果可对应到程序内部路径，便于缺陷定位分析；

(3) 相比于黑盒测试，能够更好地保障测试用例设计的完整性，提高软件功能覆盖率；

(4) 能够减少需求或设计的不详细或不完整对测试带来的不利影响。

灰盒测试的不足之处主要在于：

(1) 相比于黑盒测试，所需投入的测试时间多 20%～40%；

(2) 相比于黑盒测试，需要测试人员清楚系统内部构成及其协作关系，要求更高；

(3) 对程序代码覆盖率而言，不如白盒测试细致和深入。

## 3.4 动态测试技术分析

经过多年的发展，软件测试技术在总体上已经比较成熟，无论是静态测试技术还是动态测试技术，也不管是白盒测试、黑盒测试或者灰盒测试，都有了一些好用的测试方法。当然，这些方法各有优缺点，任何一个都不能替代另一种或者被另一种方法所替代，也正体现出了客观世界问题解决的哲学思想。

一般地，如果对被测对象认知很少，不了解软件内部结构，只关注外部的变化，比如外部输入、外部作用或被测软件所处的条件以及软件输出结果，要完成软件测试，采用黑盒测试方法；如果对被测对象非常熟悉，了解其内部结构，就可以采用白盒测试方法；而处于上述两种状态之间时，则可以采用灰盒测试方法。

另外，就软件测试级别来说，一般地，在单元测试阶段，以白盒测试方法为主；在集成测试阶段，可使用黑盒测试、白盒测试相结合的方法，或者灰盒测试方法；在集成测试之后，比如确认测试或系统测试时，主要采用黑盒测试方法。

对于某项具体的测试方法，可从如下方面进行评价。

(1) 测试用例对被测对象的覆盖程度。采用某种测试方法设计的测试用例对被测对象的覆盖程度越高，遗漏缺陷的风险就越低。

(2) 测试用例的冗余程度。测试用例的冗余程度越大，测试效率越低。每种测试方法的实施实际上都是对被测对象建模的过程，建立的模型通常都是被测对象的简化，生成的测试用例很可能存在冗余，导致用例数量多而缺陷发现率低。

(3) 测试用例集的规模大小。在满足测试效果的前提下，采用某测试方法得到的测试用例的数量越少，测试的工作量一般就越小，测试效率就越高。

（4）测试用例的缺陷定位能力。测试用例都是对应某种类型的缺陷而设计的，比如输入条件边界、异常数据处理等，好的测试方法能够在测试用例失败时，快速隔离和定位导致测试用例失败的缺陷。

（5）测试用例设计的复杂程度。测试用例的设计越简单，对测试人员的经验依赖性越低，设计测试用例所需的工作量和花费就越小，相应的测试方法越好。

CHAPTER 4 第4章

# 单元测试

按照软件研制阶段，软件测试分为单元测试、集成测试、配置项测试、系统测试和验收测试。同时，每个阶段的测试又可能包含回归测试。从动态测试角度出发，单元测试是最早开始的测试。另外，对于关键等级较高的软件，单元测试贯穿整个软件开发生命周期。因为，在后续的软件开发和测试活动中，当软件发生变更后都需要进行相应的单元测试。

在实际的软件开发项目中，存在着忽视单元测试的情况。开发人员完成编码后总是会希望尽快进行软件集成工作，以便能够实现完整的软件系统。而实际情况往往事与愿违，在后续的软件集成、软件配置项测试、软件系统测试，甚至到软件试运行过程中不断出现问题。问题有时难以定位，需要花费大量的人力、时间查找问题、解决问题，使研制进度无法保证，软件质量也让人没有信心。因此，软件开发活动中应尽早地开展软件单元测试，尽可能早地发现软件中存在的错误，从而降低软件成本，保证开发进度。

## 4.1 概述

### 4.1.1 单元测试的定义

单元测试是对软件组成的最小单元进行的测试。一般来说，软件单元是软件设计中的最小单元，一个最小单元应有明确的功能、性能和接口，并且可清晰地与其他单元划分开。根据软件开发使用的编程语言不同，软件单元的划分也不尽相同，单元的选取可参考如下原则：

（1）在结构化编程语言，例如C语言的程序中，一般认为一个函数就是一个单元；

（2）在面向对象编程语言，例如C++、Qt、Java程序中，可以将一个类作为一个单元；

（3）在可视化编程环境下的面向对象语言，例如Visual Basic、Visual C++或C#程序中，可以将一个窗体、组件作为一个单元。

### 4.1.2 单元测试的目的

有的观点认为软件测试的目的只在于发现软件错误。其实，这种观点是片面的。就软件单元测试而言，单元测试的目的是验证每个软件单元是否满足软件详细设计说明的各项要求，同时，其目的也包括发现软件单元可能存在的错误。根据现在软件工程的要求，单元

测试的主要目的包括：

(1) 验证代码是否与软件设计一致；

(2) 发现软件代码是否存在错误；

(3) 跟踪软件的实现；

(4) 复核软件设计对软件需求的实现情况。

### 4.1.3 单元测试的重要性

目前软件研发机构对软件测试的认识有了较大幅度的提升，但基于经费、进度、认识等多方面的原因，未开展全面的软件单元测试。在开发人员中，对单元测试的认识问题主要表现在以下 3 个方面。

1) 软件代码已通过调试，没必要再进行单元测试

不少开发人员认为，他们开发的软件一直不停地调试，通过调试能够正常运行的软件应该不存在错误。因此，不需要进行单元测试。但实际上，真正的软件系统是非常复杂的，仅仅通过调试是无法发现所有问题的。Humphry 的研究发现：

(1) 普通软件工程师的平均缺陷引入率为 100 个/KLoc；

(2) 受过 PSP 训练的软件工程师的平均缺陷引入率为 50 个/KLoc。

上述数据表明，不论是否受过良好的培训，不通过正式的测试，代码中会存在一定的问题。

另外，调试和测试之间从目的上就存在着很大的不同，这也是调试无法替代测试的根本原因。调试与测试在目标、方法和思路上都有所不同，两者之间的区别主要表现在以下 5 个方面：

(1) 测试从一个已知的状态开始，使用预先定义的过程，有预期的结果，调试从一个未知的状态开始，结束的过程不可预知；

(2) 测试过程可以进行设计，进度可预先估计，调试无法提前设计，持续时间也无法控制；

(3) 测试的目标是尽可能多地发现软件中的错误，调试的目的是为了解决缺陷，找出原因的一个思维分析过程；

(4) 测试能由非开发人员进行，调试必须由开发人员完成；

(5) 测试可以在没有源代码的情况下进行，调试时必须有源代码。

调试更多的是使程序能够正常地运行，而测试不仅在于证明开发人员正确地实现了设计，同时还要发现软件中是否存在问题。因此，测试能够更有效地发现软件问题，更全面地保证软件的质量。

2) 单元测试绩效不高

在很多情况下，软件开发人员在完成编码、调试后更愿意直接进行软件集成，往往觉得进行正式的单元测试工作量大，会浪费许多时间。而实际情况是，软件开发人员在完成调试后，或者进行非正式的单元测试后，直接进行软件的集成，这样造成集成过程中软件存在许多的问题，甚至有些非常低级的错误导致软件无法运行。另外，将花费大量的时间跟踪、分析、修复这些问题。尽管这些问题有的是非常低级的错误，也可能造成软件集成时花费更多的时间解决这些问题。

许多研究表明，在生命周期的早期开展软件测试，发现解决问题的成本比后期发现解决问题的成本要低很多。缺陷发现的越晚，修复它所需要的费用就越高，因此应该尽早地开展测试以发现和解决问题。表 4-1 是软件开发不同阶段修复问题所需的费用。

**表 4-1　各阶段问题修复所需费用**

| 纠错阶段 | 倍　数 |
|---|---|
| 需求分析至软件设计阶段 | 1 |
| 编程至单元测试阶段 | 5 |
| 部件至集成测试阶段 | 15 |
| 配置项至系统测试阶段 | 40 |
| 早期用户试用或 β 测试阶段 | 280 |
| 正式发布之后 | 560 |

与后续集成测试、配置项测试和系统测试相比，单元测试所需要的测试环境更容易获取，而且可重用性更好。因此，从历时长久的集成测试，或环境要求较高的配置项测试和系统测试来讲，单元测试的绩效是较高的。

Capers Jones 和 McGraw-Hill 给出了基于一个功能点单元测试、集成测试、系统测试和外场测试所需要的时间，如图 4-1 所示。单元测试的成本比集成测试、系统测试和外场测试都低。

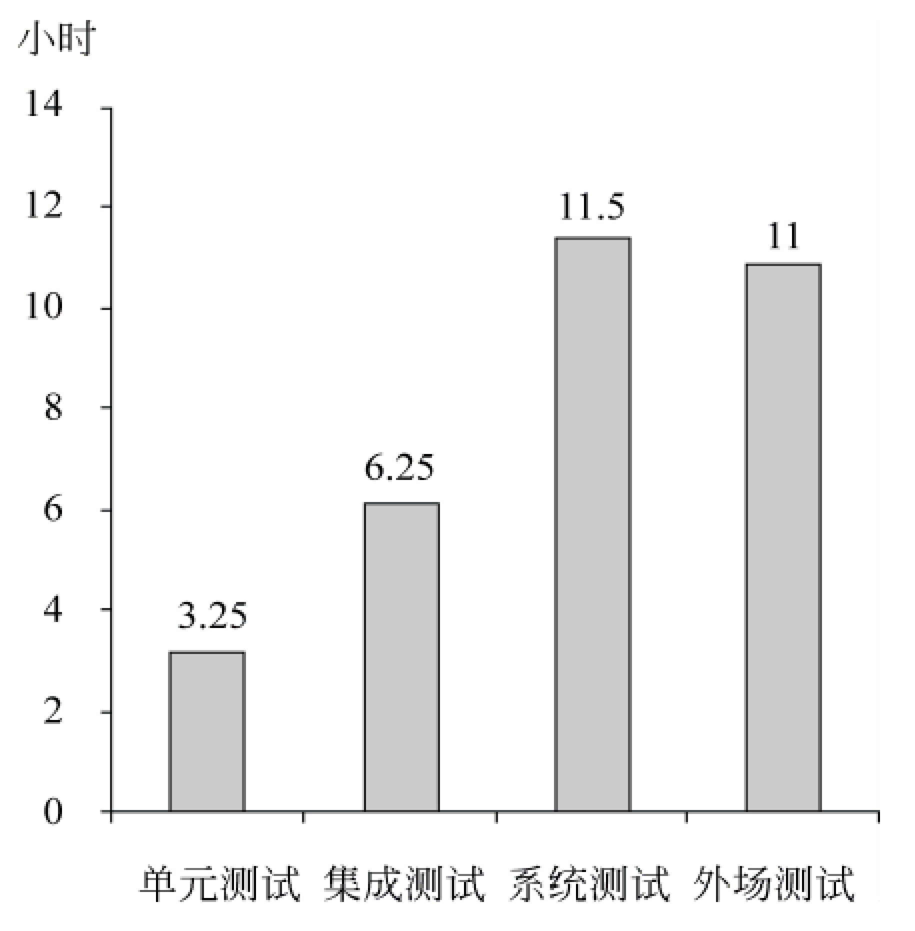


**图 4-1　各级别测试时间效率**

3）后续测试会发现 Bug，没必要进行单元测试

在前面的论述中已经对该问题进行了一些说明。单元测试是开展软件研发后续工作的基础，若不开展单元测试，软件中的许多低层次问题会带入后续的软件集成中，将会造成在软件集成过程中不停地解决本应在单元测试阶段发现和解决的问题。而且在软件集成阶段对问题的定位也将变得困难，将花费更多的时间。另外，更重要的是集成、集成测试、配置项测试等后续工作关注的重点不同，可能导致会忽略许多问题，将未发现的问题遗留至最后的软件产品中。

综上所述，单元测试是软件研制阶段中较为基础的测试。因此，单元测试的效果将直接影响到软件质量。单元测试的重要性可以从以下 3 个方面体现。

（1）进度方面。如果做好了单元测试，系统集成、配置项测试、系统测试和联调试验过程将会比较顺利，从而保证软件开发的进度；相反地，由于因为时间或人为原因不做或随便应付的单元测试，将导致在集成，以及后续工作中遇到种种问题，甚至因为一些非常低级的程序错误浪费许多时间，延误软件研制进度。

（2）效果方面。从大量的软件研制项目分析，单元测试的效果非常显著。首先，做好单

元测试，在做后续的集成测试、配置项测试和系统测试时就比较顺利。其次，在单元测试过程中能发现一些深层次的问题，还会发现一些在集成测试、配置项测试和系统测试中难以发现的问题。

(3) 成本方面。在单元测试时，某些问题很容易发现，修改所花费的成本也比较低。如果这些问题在后期的测试中发现，修改所花费的成本将成倍增加。原因在于，首先这些问题的定位难，其次这些问题的修改可能造成的影响不容易分析清楚，将使软件存在问题隐患；最后将带来多个层次上的回归测试。

从上述3个方面分析，可以说明单元测试是保证产品质量的重要环节，因此必须深刻认识到软件单元测试的重要性，重视软件单元测试，防止由于不做单元测试或随便应付而引起的后期大量时间及成本的消耗。

对于如何使单元测试的工作量与测试成效方面达到平衡，需要项目根据测试人员特点和软件单元的安全关键等级来确定。

## 4.2 单元测试原则

在进行软件单元测试时，应该遵循以下原则。

(1) 单元测试应该依据软件详细设计进行。

(2) 在对软件单元进行动态测试前，应对软件单元的源代码进行静态测试。静态测试的内容一般包括：代码和设计的一致性、代码执行标准的情况、代码逻辑表达的正确性、代码结构的合理性以及代码的可读性等。

(3) 单元测试应覆盖软件设计文档中规定的软件单元的所有功能要求。

(4) 应对单元的健壮性进行测试。

(5) 应对单元的内存使用率、响应时间等进行测试。

(6) 测试用例的输入应至少包括有效等价类值、无效等价类值和边界数据值，既要对正常的处理路径进行测试，也要对异常处理路径进行测试。

(7) 对于安全关键等级较高的软件(例如A、B级软件)，单元测试的语句、分支覆盖率均应达到100%。对于用高级语言编制的A、B级软件，还需进行修正的条件判定覆盖(MC/DC)100%测试。另外，应对没有覆盖到的语句和分支进行分析，说明其未被覆盖的原因。

(8) 应对单元的输出数据及其格式进行测试。

## 4.3 单元测试环境

单元测试的重要特点就是需要开发一定的桩模块和驱动模块，这是单元测试环境区别于其他阶段测试的重要标志。

由于软件单元往往不是一个能够独立运行的程序，所以在进行单元测试时必须开发测试用的桩模块和驱动模块。驱动模块将测试数据注入要测试的软件单元，替代调用被测软件单元的角色，同时驱动模块需要记录相关结果。桩模块的作用是替代被测单元需要调用

的软件单元，需要实现与被测单元的接口，模拟相应的操作，提供处理结果的记录。

建立单元测试环境的主要工作包括：

（1）开发必须的桩模块和驱动模块；

（2）准备单元测试数据；

（3）获取单元测试工具，包括购置和开发测试工具。

测试环境需要保证能够开展动态测试外，还应能够支持测试结果记录，测试覆盖率记录等。如图 4-2 所示单元测试环境，当被测软件单元需要调用多个模块才能完成单元测试时，需要开发桩模块模拟被调用的软件单元，如图中桩模块 1～$n$；若被调用的软件单元已通过测试时，可直接使用而不必开发桩模块，如图中软件单元 1～$m$。

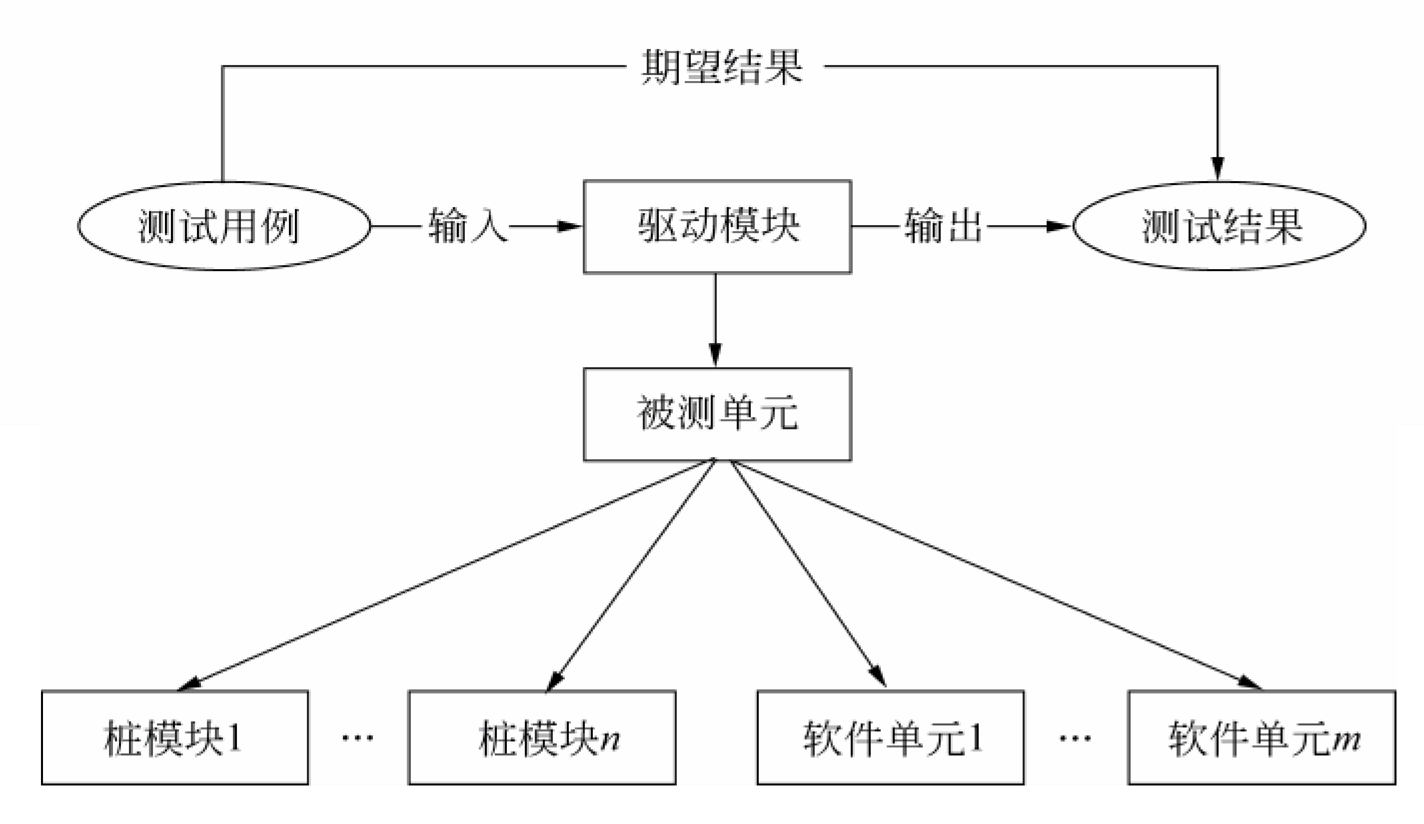



**图 4-2　单元测试环境示意图**

驱动模块和桩模块的设计原则包括：

（1）设计驱动模块和桩模块时应考虑测试用例执行所需满足的环境因素，例如前置条件、后置条件等；

（2）应充分考虑驱动模块和桩模块的复用性；

（3）桩模块的设计应保证功能上与其替代的软件单元的一致性；

（4）尽量使测试数据与测试程序分离，提高测试数据、测试程序的灵活性和重用性。

在建立单元测试环境时，驱动模块的功能应满足如下要求。

（1）应可以接收测试数据输入。输入可以包括人工输入、数据文件输入等，测试数据不仅包括注入被测软件单元的数据，还可包括期望结果数据。

（2）将测试数据注入被测软件单元。一般情况采用调用被测单元的方式进行，利用参数传递将输入数据注入被测单元。

（3）记录和输出测试结果。

在建立单元测试环境时，桩模块的功能应满足如下要求。

（1）正确地完成被模拟软件单元的基本功能。这里所谓的完成基本功能并非真正实现被模拟软件单元的功能，而是简单地按照测试用例的需要，将调用被模拟软件单元的结果返回给被测软件单元。

（2）能够被正确调用，在参数个数、类型、顺序等方面与被模拟软件单元一致。

（3）有返回值。若被模拟软件单元有返回值，应根据测试用例的要求返回被模拟软件

单元应有的返回值。

桩模块和驱动模块开发的工作量较大,有些还较为复杂,此时可以根据实际情况采取相应的单元测试策略。

# 4.4 单元测试策略

选择合适的单元测试策略,对提高单元测试效率有较为重要的影响。根据软件设计体现的软件各单元的相互关系和开发的进度,可采用自顶向下、自底向上和独立的单元测试策略,但大多数情况下,采用综合上述 3 种策略的方法开展单元测试。

## 4.4.1 自顶向下

自顶向下单元测试策略是将顶层单元所调用的软件单元作为桩模块,进行顶层模块的测试,随后对第二层单元进行测试,并使用顶层已测试过的单元作为驱动模块。以此类推完成所有单元的测试,如图 4-3 所示。图中第二层的 B2 单元测试时,可将 A1 作为驱动模块。

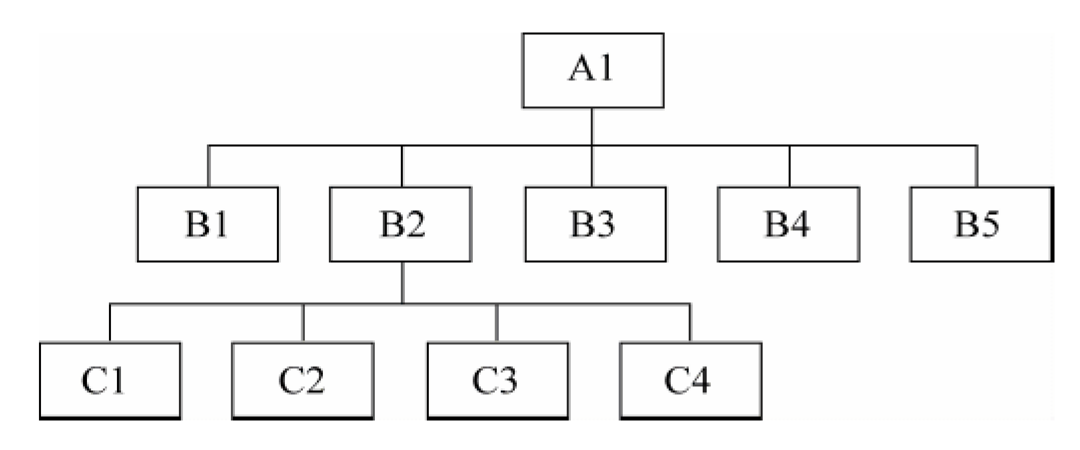



图 4-3 自顶向下测试策略示意图

该测试策略的优点主要有:

(1) 单元测试的顺序与软件设计的顺序一致;

(2) 可以使设计、编码、测试同步进行;

(3) 减少开发驱动模块的工作;

(4) 可以完成部分集成测试工作。

该测试策略的缺点主要有:

(1) 低层单元的测试须等待上层单元测试完成后才可进行,缺乏并行的灵活性,测试周期较长;

(2) 随着测试的深入,越是低层次,其结构覆盖率就越难达到;

(3) 当上层单元修改时,其下层的所有单元都要被重新测试。

该策略可以节省部分桩模块、驱动模块的开发工作,但受桩模块、驱动模块的限制,测试的充分性难以保证。

## 4.4.2 自底向上

自底向上单元测试策略是先对最低层的单元进行测试,然后以最低层作为桩模块进行

上层单元测试，以此类推完成所有模块的测试。如图 4-3 所示，第二层的 B2 单元测试时，可将 C1～C4 作为桩模块。

该测试策略的优点主要有：

(1) 减少开发桩模块的工作；

(2) 可以完成部分集成测试工作；

(3) 测试用例可以针对功能进行设计，而不必依赖设计结构。

该测试策略的缺点主要有：

(1) 上层单元的测试须等待低层单元测试完成后才可进行，缺乏并行的灵活性，测试周期较长；

(2) 随着测试的深入，越是高层次，其结构覆盖率就越难达到；

(3) 当低层单元修改时，其上层的所有单元都要被重新测试；

(4) 单元测试需要等所有单元设计、编码完成后才能进行，无法进行并行工作。

该方法在开发进度不紧张时是较好的选择，但由于该策略是面向功能的测试，很难满足结构覆盖的要求。

### 4.4.3　独立单元

独立单元测试策略是指不考虑每个单元与其他单元之间的关系，为每个模块设计桩模块和驱动模块，对每个单元进行独立的单元测试。

该测试策略的优点主要有：

(1) 简单、易操作；

(2) 由于各单元不存在依赖性，可并行开展工作；

(3) 可以达到较高的覆盖率；

(4) 可通过增加人员提高单元测试效率。

该测试策略的缺点主要有：

(1) 需要大量的桩模块、驱动模块开发工作；

(2) 需要详细的结构设计信息；

(3) 无法兼顾集成测试工作。

该策略是较好的、也是较为常用的单元测试策略。

综上所述 3 种单元测试策略，为了减少桩模块、驱动模块开发的工作量，可以考虑综合上述 3 种策略开展单元测试工作。最终选择哪种测试策略，需要根据软件的结构特点和软件单元开发的进度情况进行综合考虑。

## 4.5　单元测试内容

单元测试的目的是验证代码是否满足设计的要求，并发现在编码过程中引入的错误。单元测试的主要内容包括：功能测试、性能测试、接口测试、局部数据结构测试、边界条件测试、独立路径测试和错误处理测试。

### 4.5.1 功能测试

软件单元测试的功能测试是对软件设计文档规定的软件单元的功能进行验证。针对软件设计文档分配给软件单元的每一项功能进行测试，确认已实现的功能是否满足软件设计文档的要求。

### 4.5.2 性能测试

软件单元测试的性能测试是对软件设计文档规定的软件单元的性能进行验证。例如精度、时间、容量，应测试软件单元所实现的性能指标是否满足设计要求。

### 4.5.3 接口测试

软件单元测试的接口测试是验证进出单元的数据流是否正确，接口测试是单元测试的基础。对单元接口数据流的测试必须在其他测试之前进行。

针对单元接口进行的测试，主要涉及如下方面的内容：

（1）调用被测单元时的实际参数与该单元的形式参数的个数、属性、量纲、顺序是否一致；

（2）被测单元调用下层单元时，传递给下层单元的实际参数与下层单元的形式参数的个数、属性、量纲、顺序是否一致；

（3）是否修改了只作为输入值的形式参数；

（4）调用内部函数的参数个数、属性、量纲、顺序是否正确；

（5）被测单元在使用全局变量时是否与全局变量的定义一致；

（6）在单元有多个入口的情况下，是否引用了与当前入口无关的参数；

（7）常数是否当作变量来传递；

（8）输入/输出文件属性的正确性；

（9）OPEN 语句的正确性；

（10）CLOSE 语句的正确性；

（11）规定的输入/输出格式说明与输入/输出语句是否匹配；

（12）缓冲区容量与记录长度是否匹配；

（13）文件是否先打开后使用；

（14）文件结束条件的判断和处理的正确性；

（15）输入/输出错误是否检查并做了处理以及处理的正确性。

### 4.5.4 局部数据结构测试

在单元测试中，必须测试单元内部的数据能够保持完整性、正确性，包括内部数据的内容、格式及相互关系不发生错误。单元局部数据结构测试内容重点应考虑：

（1）局部数据的完整性，包括内容、格式及相互关系；

（2）数据类型及其说明的正确性和一致性；
（3）变量名的正确性，例如变量名是否有拼写错误或缩写错误；
（4）变量使用的正确性，例如未赋值或未初始化就使用变量；
（5）初始值或缺省值的正确性；
（6）是否有下溢、上溢或地址错误；
（7）全局数据对软件单元的影响。

### 4.5.5　边界条件测试

在对单元进行边界条件测试时，应采用边界值分析方法来设计测试用例，测试与边界值相关的数据处理是否正确。边界测试主要检查下面内容：
（1）处理 $n$ 维数组的第 $n$ 个元素时是否出错；
（2）在 $n$ 次循环的第 0 次、第 1 次、第 $n$ 次是否有错误；
（3）运算或判断中取最大和最小值时是否有错误；
（4）数据流、控制流中刚好等于、大于、小于确定的比较值时是否出现错误等。

### 4.5.6　独立执行路径测试

单元测试中，最主要的测试是针对路径的测试，在测试中应对模块中每一条独立路径进行测试。需要特别注意的测试内容主要包括以下内容。
（1）计算错误，包括：
① 算术优先级不正确；
② 混合类型的运算不正确；
③ 初始化不正确；
④ 算法不正确；
⑤ 运算精确度不满足精度要求；
⑥ 表达式的符号表示不正确。
（2）比较和控制流错误，包括：
① 不同数据类型的比较；
② 不正确的逻辑运算符或优先次序；
③ 因浮点运算精度问题而造成的比较不相等；
④ 关系表达式中不正确的变量和比较符；
⑤ 错误或不可能的循环终止条件；
⑥ 循环变量修改不正确。

### 4.5.7　错误处理测试

良好的软件设计应该考虑软件投入运行后可能发生的错误，并在程序中采取相应的措施。错误处理测试的重点是检验如果模块在工作中发生了错误，其中的出错处理是否有效。

错误处理常见问题主要有：

（1）所提供的错误描述难以理解；

（2）所提供的错误描述信息无法确定引发错误的位置或原因；

（3）显示的错误提示与实际错误不一致；

（4）对错误条件的处理不正确；

（5）对可能发生的错误未进行异常处理。

## 4.6 单元测试方法

单元测试的依据是软件详细设计，测试人员应根据详细设计说明和源程序，了解软件单元的 I/O 条件和模块的逻辑结构。一般情况下，应首先采用静态测试技术进行测试，然后根据静态测试的情况对软件单元进行动态测试，检验软件单元是否完成了规定的功能和非功能需求。

### 4.6.1 静态测试

静态测试一般采用代码审查、代码走查、静态分析。

**1. 代码审查**

代码审查是检查代码和设计的一致性、代码执行标准的情况、代码逻辑表达的正确性、代码结构的合理性以及代码的可读性。代码审查应根据所使用的语言和编码规范确定审查所用的检查单，检查单的设计应经过评审并得到相关方的确认。相关方一般应包括软件研发任务和软件测试工作的交办方。代码审查一般需使用工具完成。审查内容包括（以结构化为例）：

（1）格式；

（2）程序语言的使用；

（3）数据引用；

（4）数据声明；

（5）计算；

（6）比较；

（7）入口和出口；

（8）存储器使用；

（9）控制流；

（10）参数；

（11）逻辑和性能；

（12）维护性和可靠性。

**2. 代码走查**

代码走查是由测试人员组成小组，准备一批有代表性的测试用例，模拟计算机沿程序的

逻辑运行测试用例，查找被测软件缺陷。代码走查应：

（1）由测试人员集体阅读讨论程序；

（2）用“脑”执行测试用例并检查程序。

代码走查和代码审查的区别见表 4-2。

**表 4-2 代码走查和代码审查的区别**

| | 代码走查 | 代码审查 |
|---|---|---|
| 目标 | 逻辑错误 | 代码标准规范 |
| 准备 | 通读设计，编写用例 | 工具，检查单 |
| 形式 | 会议 | 非会议 |
| 参与人员 | 开发人员、测试人员 | 测试人员 |
| 主要技术方法 | 相关人员用“脑”执行测试用例 | 按检查单要求设置工具中检查条款，使用工具进行检查，并对工具出具的检查结果进行分析 |
| 文档 | 代码走查报告 | 代码审查报告 |

**3. 静态分析**

静态分析包括控制流分析、数据流分析、接口分析和表达式分析。一般情况下，静态分析需要使用相应的工具。测试人员将工具分析得到的结果与设计文档进行比较，查找实现与设计的不一致。

单元测试静态测试方法的详细内容见第 2 章的描述。

## 4.6.2 动态测试

完成静态测试后，还需要开展动态测试。动态测试是以测试数据为输入运行程序并全面分析输出以发现错误的过程。动态测试方法分为白盒测试方法和黑盒测试方法。

**1. 白盒测试**

白盒测试方法包括逻辑测试、数据流测试、程序变异、程序插桩、域测试和符号求值等。

一般情况下，单元测试对逻辑测试都会有较为明确的要求。逻辑测试是测试程序逻辑结构的合理性、实现的正确性。逻辑测试应由测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。逻辑测试一般需进行：

（1）语句覆盖；

（2）判定覆盖；

（3）条件覆盖；

（4）条件组合覆盖；

（5）路径覆盖。

**2. 黑盒测试**

黑盒测试方法主要用于软件单元的功能和性能方面的测试，单元测试中黑盒测试常用

的方法和技术包括：

（1）等价类划分法；

（2）边界值分析法；

（3）错误推测法；

（4）因果图法。

在进行功能测试时，需要考虑使用正常数据、边界数据和异常数据来进行测试。另外，还需要考虑接口测试等。

单元测试动态测试方法的详细内容见第 3 章的描述。

## 4.7 单元测试用例设计

对单元测试用例的设计就如同选择单元测试策略一样，不能只选择一种单一的策略，而是综合考虑测试效率，将 4.4 节中描述的 3 种测试策略综合考虑，以便获取最佳的测试效率。单元测试用例的设计也是如此，不能仅参考具体的代码，也不能仅将单元作为一个黑盒来进行测试用例设计。也就是说，不仅要测试软件单元是否做了它应该完成的事，同时还要测试软件单元是否有未赋予它的职能。因此，单元测试时应了解软件单元的内部结构，依据软件详细设计文档测试软件单元是否实现了应当做的，同时又没有多余的代码。

单元测试用例设计可以从以下 5 个方面考虑。

（1）为系统运行设计用例。第一个单元测试用例一般应覆盖该软件单元的主要功能，主要基于两个目的：

① 证明单元已具备开始测试的条件；

② 验证单元测试环境的可用性。

常用的测试用例设计方法包括：

① 规范导出法；

② 等价类划分。

（2）正向测试用例设计。测试人员应该依据软件详细设计文档设计单元测试用例，正向测试用例的作用就是验证详细设计文档中所规定的功能、性能指标是否实现。可使用等价类划分方法设计测试用例。测试用例设计时，应注意需要覆盖所有的功能、性能要求。

（3）逆向测试用例设计。逆向测试就是验证被测软件单元没有做它不应该做的事情。在设计测试用例前应对软件单元进行静态分析，比较静态分析结果与软件详细设计文档是否一致，验证软件实现是否正确。在此基础上，可使用的测试用例设计方法包括：

① 错误猜测法；

② 边界值分析法。

（4）特殊要求的测试用例设计。在对安全关键等级较高的软件进行单元测试时，需要对软件单元的性能、安全性、保密性等设计测试用例。常使用的方法是规范导出法，以及在安全性分析的基础上进行用例设计。

（5）覆盖率测试用例设计。通过上述（1）～（4）进行的测试用例设计已经可以开展单元测试了，但为了达到软件单元测试的充分性要求，需要满足一定的覆盖率指标。当执行完上述测试后，若未能达到覆盖率指标要求，还需要分析覆盖率情况，确定未覆盖的原因，以便有

针对性地补充设计测试用例。当确实因用例设计不充分时，可根据分析结果补充相应的测试用例。未达到覆盖率的原因可能有：

① 不可能注入的条件；
② 不可达或冗余代码；
③ 不充分的测试用例。

## 4.8　单元测试过程

单元测试依据软件设计文档，按照如图 4-4 所示过程开展测试，单元测试过程分为以下 4 个阶段。

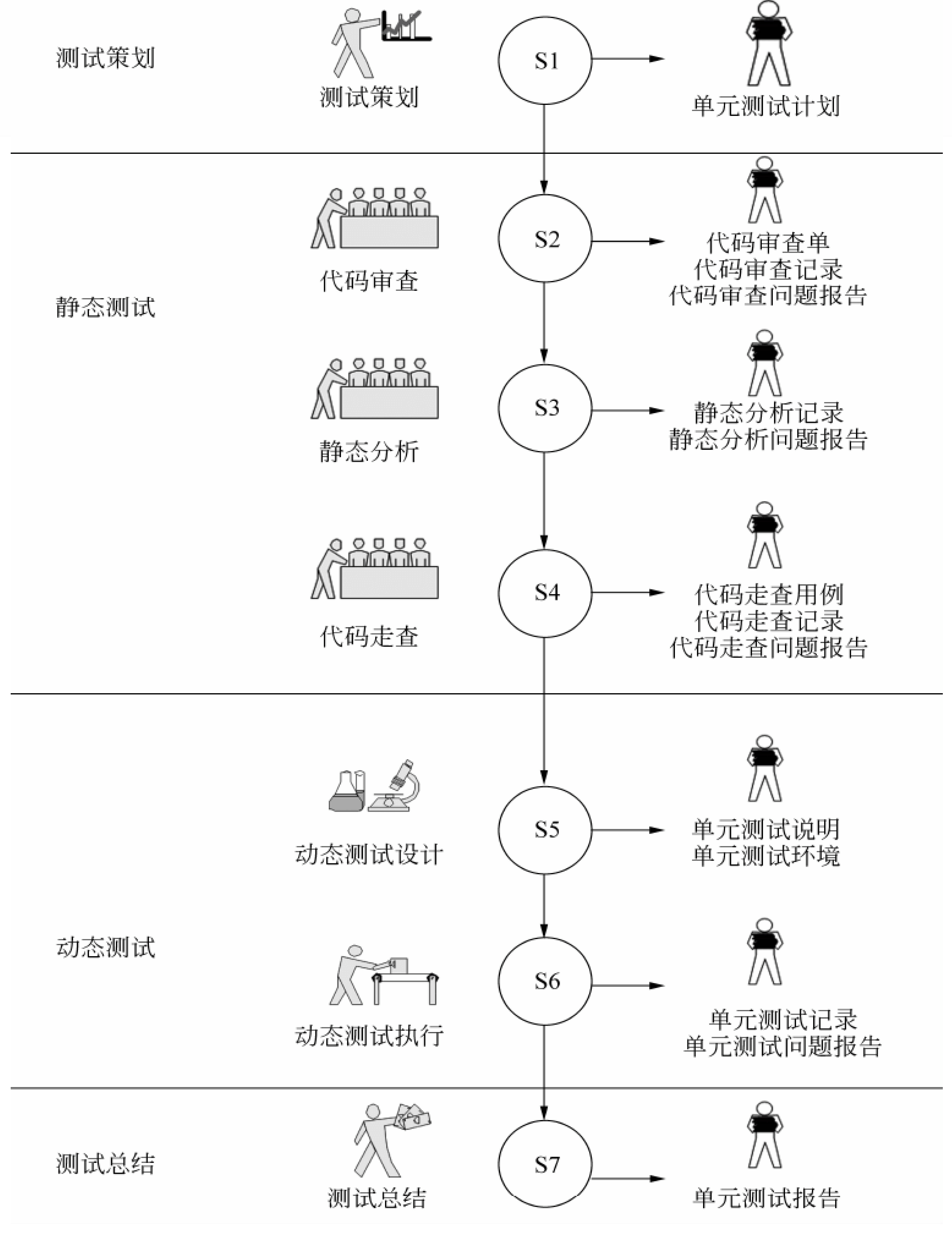



图 4-4　单元测试过程

（1）测试策划。依据软件设计文档进行单元测试策划，制定单元测试计划。

（2）静态测试。依据单元测试计划开展软件单元的静态测试，并对静态测试问题进行回归测试。

（3）动态测试。在静态测试的基础上，按照单元测试计划规定的内容开展动态测试用例的设计，完成桩模块和驱动模块的设计，准备动态测试数据，并按照测试计划、测试说明完成动态测试，记录测试结果，完成相应的回归测试。

（4）测试总结。分析静态测试和动态测试结果，分析测试的充分性，判断是否通过单元测试。

## 4.8.1 测试策划

单元测试策划应尽早进行，一般情况下，可以在软件设计初步完成时就开始进行策划。单元测试存在的问题主要表现在以下 6 个方面：

（1）对单元测试的重要性认识不足；

（2）单元测试测试开始较晚，大部分受资源限制，在编码结束后才开始进行单元测试；

（3）单元测试用例设计不够全面，常见的问题表现在根据已实现的代码进行用例设计，而忽略了依据设计进行测试的要求；

（4）没有按照一定的原则选择被测软件单元，有时为非常简单的单元编写驱动模块和桩模块，浪费了大量的时间；

（5）在构建测试环境时，缺乏可复用性的考虑；

（6）没有考虑使用有效的工具开展单元测试。

因此，为了达到质量、进度、效率的平衡，应在单元测试策划时考虑如下因素：

（1）软件的质量要求；

（2）软件研制的进度安排；

（3）软件单元的关键等级；

（4）测试资源的限制。

单元测试策划的内容包括以下内容。

（1）按照软件设计和软件质量要求确定软件单元测试的需求。包括：

① 需要进行测试的软件单元名称、标识；

② 需要测试的内容；

③ 提出每个单元的测试方法，包括驱动模块和桩模块的要求；

④ 提出每个单元测试的充分性要求；

⑤ 测试终止条件；

⑥ 优先级；

⑦ 对软件设计文档的追踪关系。

（2）提出单元测试环境要求，包括测试工具等。

（3）提出单元测试人员安排。

（4）安排单元测试的进度计划。应依据测试需求、测试环境、人员等情况，制定合理可行的软件单元测试进度计划。

(5) 制定单元测试通过的准则。单元测试通过的准则如下：

① 软件单元功能与设计一致；

② 软件单元接口与设计一致；

③ 能够正确处理输入和运行中的异常情况；

④ 单元测试发现的问题得到修改并通过回归测试；

⑤ 达到了覆盖率的要求；

⑥ 单元测试报告通过评审。

单元测试策划完成时应编写软件单元测试计划。单元测试计划是否合理可行、是否满足充分性要求是十分值得关注的，因此应对软件单元测试计划进行评审。评审的主要内容包括：

(1) 测试人员组成合理、分工明确；

(2) 明确单元测试策略，测试通过的准则；

(3) 根据研制任务书、软件设计文档和其他软件质量要求明确需要进行测试的软件单元；

(4) 测试资源满足单元测试任需求；

(5) 对每个被测单元提出圈复杂度(即 McCabe 复杂性度量值)的度量要求；

(6) 对每个软件单元的扇入、扇出数提出分析和统计要求；

(7) 对软件单元源代码注释率(即有效注释行与源代码总行的比率)提出分析检查要求；

(8) 对软件可靠性、安全性设计准则和编程准则提出检查要求(要求关键等级较高的软件应落实全部强制类编程准则)；

(9) 对源代码与软件设计文档提出一致性的分析、检查要求；

(10) 对有特殊要求的软件单元，进行特殊测试，例如占用空间、运行时间、计算精度等测试要求；

(11) 对于重要的执行路径，提出路径测试要求；

(12) 提出边界测试要求；

(13) 明确语句覆盖率要求；

(14) 明确软件测试分支覆盖率要求；

(15) 明确修正的条件判定覆盖(MC/DC)的覆盖要求；

(16) 明确提出单元测试的终止条件；

(17) 建立单元测试项(条目)到软件设计之间追踪关系。

### 4.8.2 静态测试

对软件单元的静态测试常用方法包括以下 3 种：

(1) 代码审查；

(2) 静态分析；

(3) 代码走查。

但是，对于一个软件单元来说不一定 3 种方法都用，根据实际情况选择最合适的 1～2 种方

法即可。

一般情况下，软件单元的动态测试应在完成静态测试后，当所有的静态测试问题通过回归测试确认都得到解决后再进行动态测试较为妥当。

**1. 代码审查**

按照 GJB 2725A—2001《测试实验室和校准实验室通用要求》以及《军用软件测评实验室测评过程和技术能力要求》规定的代码审查内容包括：

(1) 代码和设计的一致性；

(2) 代码执行标准的情况；

(3) 代码逻辑表达的正确性；

(4) 代码结构的合理性以及代码的可读性。

代码审查应根据研制任务书中规定的开发语言和编码规范要求，确定审查所用的检查单，检查单应经过评审。

表 4-3 给出了某项目中对 C 语言程序进行代码审查时所选定的审查项。

**表 4-3 C 语言代码审查项示例**

| 序号 | 审查项 |
|---|---|
| 1 | 过程名禁止被重用 |
| 2 | 赋值类型必须匹配 |
| 3 | 符号名禁止被重用 |
| 4 | 数组下标必须是整型数 |
| 5 | 禁止在结构体定义中含有空域 |
| 6 | 禁止对常数值做逻辑非的运算 |
| 7 | 禁止声明多重标号 |
| 8 | 禁止对有符号类型使用位运算 |
| 9 | 参数必须使用类型声明 |
| 10 | 禁止枚举类型的越界使用 |
| 11 | 在过程声明中必须对参数说明 |
| 12 | 变量的使用禁止超出所定义的范围 |
| 13 | 禁止过程参数只有类型没有标识符 |
| 14 | 禁止在条件判别语句中使用赋值操作符 |
| 15 | 禁止在过程参数表中使用省略号 |
| 16 | 禁止赋值操作符与“&&”或“\|\|”连用 |
| 17 | 禁止重新定义使用 C 或 C++ 关键字 |
| 18 | 禁止位操作符带有布尔型的操作数 |
| 19 | 禁止过程/函数中参数表为空 |

续表

| 序号 | 审　查　项 |
|---|---|
| 20 | 禁止位操作符作用于布尔值 |
| 21 | 禁止在宏中使用了多个“＃”或“＃＃” |
| 22 | 字符串数组的赋值必须再所分配的空间之内 |
| 23 | 禁止定义不像函数的宏 |
| 24 | 禁止移位操作符中的右操作数为负数 |
| 25 | 禁止在宏中包含不允许的项 |
| 26 | 禁止 sizeof 操作符的操作数是外部声明的数组 |
| 27 | 禁止重新定义保留字 |
| 28 | 禁止 sizeof 的参数是数组类型的函数参数 |
| 29 | 字符类型必须标明是有符号还是无符号 |
| 30 | 禁止“&&”“\|\|”或“!”操作符的操作数不是布尔型 |
| 31 | 禁止对一个名字重新定义 |
| 32 | 禁止使用已释放了的存储空间 |
| 33 | 用 typedef 自定义的类型名禁止被重新定义 |
| 34 | 动态分配的存储空间在返回或退出活动范围时必须释放 |
| 35 | 禁止在同一个文件中有＃if 而没有＃endif |
| 36 | 禁止修改或释放字符串常量 |
| 37 | 禁止数组没有边界限定 |
| 38 | 实参与形参个数必须一致 |
| 39 | 禁止在＃include＜……＞中使用绝对路径名 |
| 40 | 主过程所在文件中禁止有未被该文件中任何过程调用的子过程 |
| 41 | 禁止结构体声明不完整 |
| 42 | static 类型的过程在其所在文件中必须被调用 |
| 43 | 禁止参数的声明形式上不一致 |
| 44 | 禁止使用禁用的过程、函数、文件或名称 |
| 45 | 函数参数中的一维数组，禁止限定为固定长度 |
| 46 | 在不能使用 extern 的文件中禁止使用 extern |
| 47 | 函数参数中多维数组，第一维禁止限定为固定长度，其他维禁止为空 |
| 48 | 禁止同一个表达式中调用多个相关函数 |
| 49 | 禁止函数、变量或常量使用不同的类型重定义 |
| 50 | 禁止 void 类型的过程用在表达式中使用 |
| 51 | 禁止在函数体内不加任何注释地使用外部声明 |

续表

| 序号 | 审　查　项 |
|---|---|
| 52 | 禁止 void 类型的变量作为参数进行传递 |
| 53 | 禁止对同一函数作多次声明 |
| 54 | 禁止实参和形参类型不一致 |
| 55 | 过程体必须用大括号括起来 |
| 56 | 函数和原型参数类型必须一致 |
| 57 | 循环体必须用大括号括起来 |
| 58 | 禁止单独使用小写的“l”或大写的“O”作为变量名 |
| 59 | else 中的语句必须用大括号括起来 |
| 60 | 禁止三字母词的使用 |
| 61 | 逻辑表达式的连接必须使用大括号 |
| 62 | 使用的八进制数必须加以注释 |
| 63 | 禁止在头文件前有可执行代码 |
| 64 | 禁止使用不起作用的语句 |
| 65 | 宏参数必须用括号括起来 |
| 66 | 函数必须有返回语句 |
| 67 | 嵌入汇编程序的过程必须是纯汇编程序 |
| 68 | 禁止返回类型说明为 void 的过程中有 retum 语句返回值 |
| 69 | 头文件名禁止使用“ 、”“\”和“/”等字符 |
| 70 | 有返回值的函数中 return 必须带有返回值 |
| 71 | 字符型数组赋值时，必须使用“\0”终止字符串 |
| 72 | 函数返回类型必须一致 |
| 73 | main 函数必须定义为“int main(void)”或“int main(int，char * [])”的形式 |
| 74 | 函数和原型返回类型必须一致 |
| 75 | 禁止条件判别成立时相应分支无执行语句 |
| 76 | 当函数返回时，禁止利用外部变量指向堆栈区而对堆栈区进行引用 |
| 77 | 在“if... else if”语句中必须使用有 else 分支 |
| 78 | 禁止使用嵌套的注译 |
| 79 | 在“if... else if”语句中禁止 else 分支无可执行语句 |
| 80 | 禁止使用不合适的循环变量类型 |
| 81 | 在 switch 语句中必须有 defult 语句 |
| 82 | 循环变量必须是局部声明的 |
| 83 | 禁止使用空 switch 语句 |

续表

| 序号 | 审　查　项 |
| --- | --- |
| 84 | 禁止对指针变量使用强制类型转换赋值 |
| 85 | 禁止 switch 语句中只包含 default 语句 |
| 86 | 禁止对指针变量赋值类型不匹配 |
| 87 | 禁止 switch 的 case 语句不是由 break 终止 |
| 88 | 枚举元素的初始化必须完整 |
| 89 | 禁止 switch 的 case 语句中无任何可执行语句 |
| 90 | 结构体变量初始化的类型必须一致 |
| 91 | 禁止当 switch 语句的表达式是枚举类型时缺少 case 分支 |
| 92 | 结构体变量初始化的嵌套结构必须和定义一致 |
| 93 | 禁止将参数指针赋值给过程指针 |
| 94 | 变量使用前必须被赋值过 |
| 95 | 禁止指针的指针超过两级 |
| 96 | 枚举类型的变量的成员必须初始化为整型 |
| 97 | 禁止将过程声明为指针类型 |
| 98 | 数组的初始化必须完整 |
| 99 | 禁止引用空指针 |
| 100 | 禁止对实数类型的量做是否相等的比较 |
| 101 | 谨慎使用指针的逻辑比较 |
| 102 | 禁止逻辑判别的表达式不是逻辑表达式 |
| 103 | 谨慎对指针进行代数运算 |
| 104 | switch 语句中的表达式禁止是逻辑表达式 |
| 105 | 禁止程序正常自行时直接从过程中跳出 |
| 106 | 逻辑判别表达式如果存在运算项,必须使用括号 |
| 107 | 禁止使用 goto 语句 |
| 108 | 对无符号数进行大于等于零或小于零的比较 |
| 109 | 禁止在非赋值表达式中使用赋值操作符 |
| 110 | 禁止两个布尔型变量进行比较 |
| 111 | 数组的使用必须保证不会出现越界 |
| 112 | 禁止枚举类型中的元素与已有变量同名 |
| 113 | 禁止对有符号类型进行移位运算 |
| 114 | 禁止局部变量与全局变量同名 |
| 115 | 对变量进行移位运算必须保证不会产生溢出 |

续表

| 序号 | 审　查　项 |
|---|---|
| 116 | 禁止参数与全局变量同名 |
| 117 | 禁止给无符号变量赋负值 |
| 118 | 禁止参数与类型或标识符同名 |
| 119 | 有符号类型的位长度必须大于等于两位 |
| 120 | 禁止在内部块中重定义已有的变量名 |
| 121 | 位的定义必须是有符号整数或无符号整数 |
| 122 | 禁止在复杂的表达式中使用 volatile 类型的变量 |
| 123 | 禁止给变量赋的值与变量的类型不一致 |
| 124 | 禁止将一个不是枚举成员的值赋给枚举变量 |

按照代码审查的要求，一般需要工具辅助完成代码执行标准情况、代码可读性等的检查，例如：

（1）对每个软件单元的圈复杂度（即 McCabe 复杂性度量值）进行分析检查；

（2）对软件单元规模（即软件单元源代码行数）进行统计；

（3）对软件单元的扇出数进行分析和统计；

（4）对软件单元源代码的注释行与源代码总行数的比率进行分析。

代码审查的记录应进行保存，对代码复杂度、规模、注释率、扇出数的审查结果进行记录的示例如表 4-4 所示。

**表 4-4　软件单元复杂度、语句数、注释数、注释率、扇出数审查结果示例**

| 编号 | 文件名称 | 编号 | 子程序名称 | 复杂度 | 语句数 | 注释数 | 注释率 | 扇出数 |
|---|---|---|---|---|---|---|---|---|
| 1 | bus. c | 1 | write_ports | 1 | 17 | 11 | 64. 71% | 0 |
| | | 2 | read_ports | 1 | 17 | 13 | 76. 47% | 0 |
| | | 3 | init_1553B | 5 | 52 | 37 | 71. 15% | 1 |
| | | 4 | ClearINT | 1 | 13 | 10 | 76. 92% | 1 |
| 2 | can. c | 5 | init_82527 | 3 | 81 | 44 | 54. 32% | 0 |
| 3 | dsptms. c | 6 | two_out_of_three | 4 | 33 | 13 | 39. 39% | 0 |
| | | 7 | self_data_DRAM | 11 | 132 | 48 | 36. 36% | 2 |
| | | 8 | self_data | 1 | 71 | 42 | 59. 15% | 0 |
| | | 9 | self_mode | 2 | 21 | 16 | 76. 19% | 3 |
| | | 10 | Normal_mode | 9 | 62 | 34 | 54. 84% | 5 |
| | | 11 | main | 5 | 33 | 17 | 51. 52% | 5 |
| | | 12 | SysInit | 1 | 43 | 39 | 90. 70% | 3 |
| | | 13 | MemInit | 9 | 52 | 23 | 44. 23% | 0 |

续表

| 编号 | 文件名称 | 编号 | 子程序名称 | 复杂度 | 语句数 | 注释数 | 注释率 | 扇出数 |
|---|---|---|---|---|---|---|---|---|
| | | 14 | c_int01 | 3 | 34 | 27 | 79.41% | 1 |
| | | 15 | c_int02 | 3 | 28 | 17 | 60.71% | 0 |
| | | 16 | c_int03 | 1 | 22 | 16 | 72.73% | 0 |
| | | 17 | c_int04 | 43 | 294 | 94 | 31.97% | 8 |
| | | 18 | c_int05 | 1 | 12 | 10 | 83.33% | 0 |
| | | 19 | c_int06 | 11 | 71 | 41 | 57.75% | 0 |
| | | 20 | ManageProc | 10 | 66 | 42 | 63.64% | 2 |
| | | 21 | DataTran | 2 | 22 | 15 | 68.18% | 0 |
| | | 22 | CANProc | 17 | 131 | 75 | 57.25% | 0 |
| | | 23 | hang_double_ram | 2 | 21 | 11 | 52.38% | 0 |
| | | 24 | TempToAscii | 3 | 41 | 18 | 43.90% | 0 |
| | | 25 | online_DRAM | 1 | 24 | 13 | 54.17% | 0 |
| 3 | dsptms.c | 26 | online | 1 | 28 | 14 | 50.00% | 1 |
| | | 27 | write_DRAM | 24 | 240 | 83 | 34.58% | 3 |
| | | 28 | packet_data | 15 | 172 | 61 | 35.47% | 2 |
| | | 29 | write_blood | 2 | 41 | 15 | 36.59% | 0 |
| | | 30 | CAN_A | 2 | 39 | 29 | 74.36% | 0 |
| | | 31 | CAN_B | 2 | 39 | 24 | 61.54% | 0 |
| | | 32 | CAN_C | 2 | 39 | 24 | 61.54% | 0 |
| | | 33 | CAN_D | 2 | 39 | 24 | 61.54% | 0 |
| | | 34 | blood_disposal | 22 | 109 | 34 | 31.19% | 1 |
| | | 35 | blood_receive | 5 | 60 | 19 | 31.67% | 1 |
| | | 36 | mem_status_self | 1 | 14 | 11 | 78.57% | 0 |
| | | 37 | PD_ASCII | 4 | 24 | 9 | 37.50% | 0 |
| | | 38 | PD_XL_validity | 2 | 23 | 9 | 39.13% | 0 |

对于代码和设计的一致性、代码逻辑表达的正确性、代码结构的合理性以及代码注释行有效性、可理解性等，需要通过人工阅读代码来完成。由于这部分的工作量较大，可选择核心模式或逻辑复杂的模块进行。

**2. 静态分析**

按照 GJB 2725A—2001《测试实验室和校准实验室通用要求》以及《军用软件测评实验室测评过程和技术能力要求》规定的静态分析内容包括：

(1) 控制流分析；

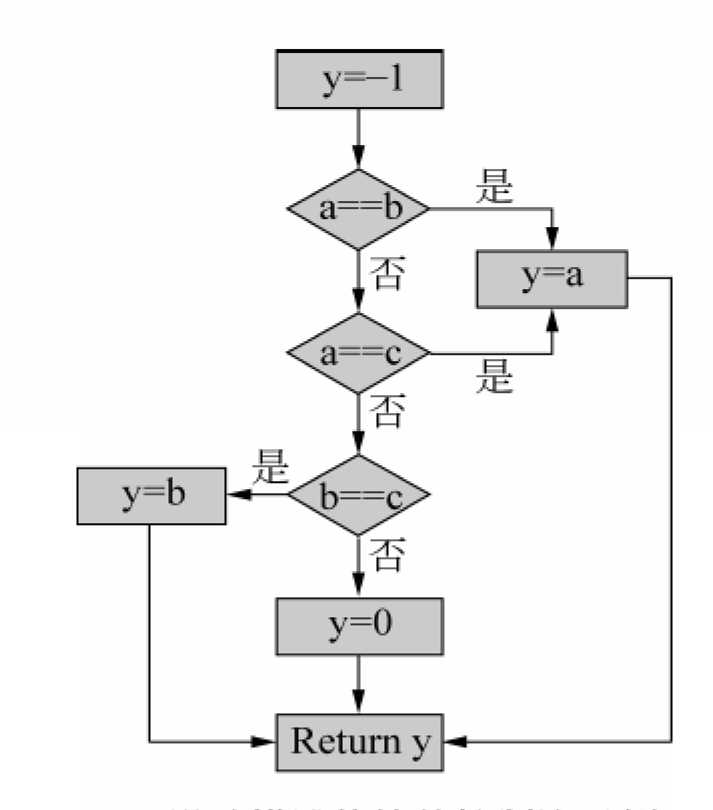
(2) 数据流分析；

(3) 接口分析；

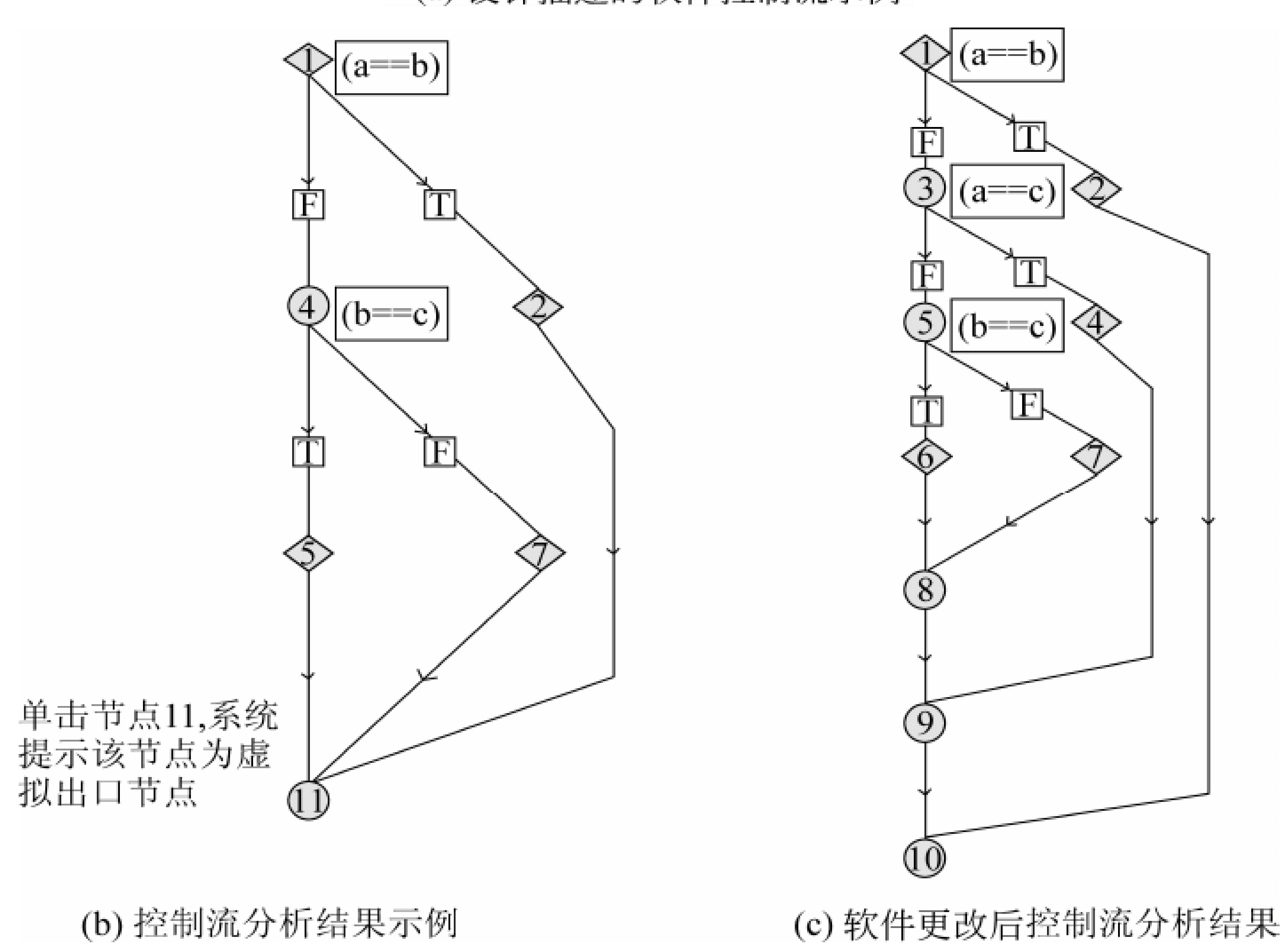
(4) 表达式分析。

静态分析需要借助工具完成，并根据工具提供的分析记录与设计文档进行比较，分析是否与设计要求一致。

下面给出用工具进行的某软件单元控制流分析的结果示例，如图 4-5 所示。其中，图 4-5(a)为设计文档中描述的控制流，图 4-5(b)为工具分析得出的软件实现的控制流，图 4-5(c)为纠错后工具重新分析得出的软件实现的控制流。




(a) 设计描述的软件控制流示例




(b) 控制流分析结果示例　　(c) 软件更改后控制流分析结果

**图 4-5　某软件单元控制流分析的结果示例**

将图 4-5(b)的结果与设计文档进行比较，发现该单元的程序实现与设计不一致，未实现一个判断分支，并违反了代码实现不允许有多个函数出口的规则，具体表现为源代码中有多个 return 返回，而详细设计说明中程序流图只有一个出口。

**3. 代码走查**

按照 GJB 2725A—2001《测试实验室和校准实验室通用要求》以及《军用软件测评实验室测评过程和技术能力要求》规定的代码走查是由测试人员组成小组，准备一批有代表性的测试用例，集体扮演计算机的角色，按程序的逻辑，逐步运行测试用例，查找被测软件缺陷。

由于代码走查需要的人员和时间都比较多，大多数软件研制项目都只选择核心、关键单元，或新开发、变更的单元进行代码走查。

代码走查的方法可以用于以下情况：

（1）当建立仿真测试环境过于复杂或无法建立；

（2）动态测试无法到达的路径；

（3）软件单元关键等级较高，需要提高测试充分性。

## 4.8.3　动态测试

完成静态测试后，应进行软件单元的动态测试。动态测试与静态测试应从不同方面验证软件单元，静态测试和动态测试无法相互替代。

软件单元测试的动态测试方法包括：

（1）等价类划分；

（2）边界值分析法；

（3）错误推测法；

（4）因果图法。

软件单元动态测试主要是依据软件设计文档、软件静态测试结果以及软件单元测试计划的要求开展如下活动：

（1）进行单元测试用例设计，并编写软件单元测试说明；

（2）根据测试设计的要求编写驱动模块和桩模块；

（3）获取测试计划中提出的单元测试工具；

（4）对单元测试环境进行验证；

（5）按照测试计划、测试说明执行动态测试，记录测试结果，提交问题报告；

（6）分析测试执行情况，审查测试是否满足充分性要求，若存在问题需要补充相应的测试用例时，应采取相应的措施补充测试用例以满足测试充分性要求；

（7）根据问题修改情况进行回归测试。

在进行软件单元的动态测试过程中会提出覆盖率要求来保证测试的充分性。对覆盖率情况，一般的单元测试工具都提供统计分析的功能。软件测试人员应记录软件单元的动态测试覆盖率情况的原始记录，原始记录示例如表 4-5 所示。

**表 4-5　分支覆盖率统计表**

| 编号 | 子程序名 | 语句覆盖数 | 语句总数 | 语句覆盖率 | 分支覆盖数 | 分支总数 | 分支覆盖率 | MC/DC 覆盖数 | MC/DC 总数 | MC/DC 覆盖率 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | write_ports | 6 | 6 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 2 | read_ports | 6 | 6 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |

续表

| 编号 | 子程序名 | 语句覆盖数 | 语句总数 | 语句覆盖率 | 分支覆盖数 | 分支总数 | 分支覆盖率 | MC/DC覆盖数 | MC/DC总数 | MC/DC覆盖率 |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | init_1553B | 34 | 34 | 100.00% | 9 | 9 | 100.00% | 9 | 9 | 100.00% |
| 4 | ClearINT | 2 | 2 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 5 | init_82527 | 134 | 134 | 100.00% | 4 | 6 | 66.67% | 4 | 6 | 66.67% |
| 6 | two_out_of_three | 14 | 14 | 100.00% | 7 | 10 | 70.00% | 7 | 10 | 70.00% |
| 7 | self_data_DRAM | 94 | 94 | 100.00% | 24 | 24 | 100.00% | 24 | 24 | 100.00% |
| 8 | self_data | 61 | 61 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 9 | self_mode | 8 | 8 | 100.00% | 4 | 4 | 100.00% | 4 | 4 | 100.00% |
| 10 | Normal_mode | 36 | 36 | 100.00% | 19 | 19 | 100.00% | 19 | 19 | 100.00% |
| 11 | main | 16 | 17 | 94.12% | 11 | 12 | 91.67% | 11 | 12 | 91.67% |
| 12 | SysInit | 30 | 30 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 13 | MemInit | 34 | 34 | 100.00% | 17 | 17 | 100.00% | 17 | 17 | 100.00% |
| 14 | c_int01 | 19 | 19 | 100.00% | 7 | 7 | 100.00% | 7 | 7 | 100.00% |
| 15 | c_int02 | 16 | 17 | 94.12% | 3 | 4 | 75.00% | 3 | 4 | 75.00% |
| 16 | c_int03 | 12 | 12 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 17 | c_int04 | 179 | 201 | 89.05% | 100 | 114 | 87.72% | 100 | 114 | 87.72% |
| 18 | c_int05 | 2 | 2 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 19 | c_int06 | 42 | 42 | 100.00% | 25 | 25 | 100.00% | 25 | 25 | 100.00% |
| 20 | ManageProc | 51 | 53 | 96.23% | 27 | 28 | 96.43% | 27 | 28 | 96.43% |
| 21 | DataTran | 11 | 11 | 100.00% | 3 | 3 | 100.00% | 3 | 3 | 100.00% |
| 22 | CANProc | 91 | 91 | 100.00% | 39 | 40 | 97.50% | 45 | 49 | 91.84% |
| 23 | hang_double_ram | 10 | 10 | 100.00% | 3 | 3 | 100.00% | 3 | 3 | 100.00% |
| 24 | TempToAscii | 20 | 25 | 80.00% | 6 | 7 | 85.71% | 6 | 7 | 85.71% |
| 25 | online_DRAM | 13 | 13 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 26 | online | 18 | 18 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 27 | write_DRAM | 145 | 159 | 91.19% | 57 | 64 | 89.06% | 57 | 64 | 89.06% |
| 28 | packet_data | 106 | 120 | 88.33% | 36 | 43 | 83.72% | 36 | 43 | 83.72% |
| 29 | write_blood | 28 | 28 | 100.00% | 3 | 3 | 100.00% | 3 | 3 | 100.00% |
| 30 | blood_disposal | 24 | 24 | 100.00% | 12 | 12 | 100.00% | 12 | 12 | 100.00% |
| 31 | blood_receive | 40 | 40 | 100.00% | 13 | 13 | 100.00% | 19 | 19 | 100.00% |
| 32 | mem_status_self | 4 | 4 | 100.00% | 1 | 1 | 100.00% | 1 | 1 | 100.00% |
| 33 | PD_ASCII | 8 | 8 | 100.00% | 10 | 10 | 100.00% | 19 | 19 | 100.00% |
| 34 | PD_XL_validity | 11 | 11 | 100.00% | 4 | 4 | 100.00% | 7 | 7 | 100.00% |
| 35 | CAN_A | 27 | 27 | 100.00% | 4 | 4 | 100.00% | 4 | 4 | 100.00% |

续表

| 编号 | 子程序名 | 语句覆盖数 | 语句总数 | 语句覆盖率 | 分支覆盖数 | 分支总数 | 分支覆盖率 | MC/DC覆盖数 | MC/DC总数 | MC/DC覆盖率 |
|---|---|---|---|---|---|---|---|---|---|---|
| 36 | CAN_B | 27 | 27 | 100.00% | 4 | 4 | 100.00% | 4 | 4 | 100.00% |
| 37 | CAN_C | 27 | 27 | 100.00% | 4 | 4 | 100.00% | 4 | 4 | 100.00% |
| 38 | CAN_D | 27 | 27 | 100.00% | 4 | 4 | 100.00% | 4 | 4 | 100.00% |
| | 合计 | 1433 | 1492 | 96.05% | 469 | 507 | 92.50% | 493 | 534 | 92.32% |

### 4.8.4 测试总结

软件单元测试完成后应对单元测试工作进行总结，以便评估软件单元测试中的问题是否得到解决，单元测试工作是否达到充分性要求。单元测试总结的内容包括以下内容。

(1) 对单元测试过程进行总结。应对单元测试策划、静态测试、动态测试过程进行总结，说明在测试策划过程、静态测试和动态测试中开展的主要工作、参与的人员和工作完成情况。

(2) 对单元测试方法进行说明。应说明单元测试所采用的测试方法与策略，并说明采用这些方法的依据。

(3) 对单元测试环境进行分析。应说明单元测试所使用的测试环境，包括测试工具、桩模块和驱动模块的情况，并对测试环境的差异性进行分析，说明测试环境是否满足单元测试的要求。

(4) 对单元测试结果进行分析。单元测试结果的分析应包括对静态测试、动态测试以及所有回归测试的情况的分析。主要内容包括：

① 测试时间；

② 测试人员；

③ 测试用例执行情况，内容包括测试用例数、通过的测试用例、未通过的测试用例、完全执行的测试用例、未完全执行的测试用例、未执行的测试用例；

④ 测试覆盖情况，包括对软件单元的覆盖情况，每个单元的覆盖情况：包括语句、分支、路径等的覆盖情况，说明是否满足测试充分性要求；

⑤ 说明单元测试过程中发现的问题，并对问题解决情况进行说明；

⑥ 若有遗留问题应说明遗留问题的处理措施；

⑦ 若有无法执行的测试应说明后续测试的计划等；

⑧ 对软件单元测试的整体情况进行评价，并提出改进的意见建议。

第5章 CHAPTER 5

# 集成测试

集成测试是在单元测试的基础上进行的，集成测试是将所有的软件单元按照软件设计逐步组装成软件配置项、子系统或系统的过程，并测试集成后的各部分工作是否协调一致，达到或实现相应技术指标及要求的活动。因此，在集成测试之前，软件单元测试应该已经完成，集成测试中所使用的对象应该是已经通过单元测试的软件单元。这一点非常的关键，因为如果不经过单元测试，那么集成测试的效果将会受到很大影响，会因为定位和纠正软件单元代码的错误增加更多的工作量，也会因为集成测试的深度问题而遗漏一些关键性的缺陷。

## 5.1 概述

### 5.1.1 集成测试的定义

集成测试是在单元测试的基础上，将通过单元测试的模块按设计要求把它们逐步组装起来，形成系统，并进行测试。

按照系统的设计，如果系统只包含一个软件配置项，那么集成测试只需要从软件单元集成为软件配置项即可。如果系统由多个配置项构成，那么集成工作将分为两步进行：一是将软件单元集成为软件配置项，称为配置项集成测试；二是将配置项（包括软件配置项和硬件配置项）集成为系统，称为系统集成测试。对于特别复杂的系统，可能包含多个子系统，每个子系统中又包含多个配置项。这种情况下，需要进行单元集成为软件配置项，配置项集成为子系统，子系统集成为系统的测试。其中，配置项集成为子系统与配置项集成为系统的方法和要求基本一致。

在大型系统的开发过程中，特别是重大工程任务中集成测试一般分为多个层次开展，按照 GJB 2786A—2009《军用软件开发通用要求》的划分，分为配置项集成和系统集成两个层次。

集成测试主要是测试软件单元或配置项集成后能否正常工作。集成测试前应完成单元测试，系统集成测试前应完成配置项测试。配置项集成测试中各软件单元应是已经通过单元测试的软件单元，系统集成测试中各配置项应是已通过配置项测试的配置项。这个条件非常重要，因为如果不经过单元测试或配置项测试，集成测试的效果将会受到很大影响，并且会大幅增加软件纠错的代价。

### 5.1.2　集成测试的目的

集成测试是一个由单元测试到系统测试的过渡测试，往往容易被忽视。造成的后果是系统测试困难重重，系统无法协调一致的运行，软件问题频发，发现的问题难以定位，遗留的软件问题隐患较多。因此，单元测试后，应进行有计划的集成测试。

当然，集成测试的目的应与软件研制的目标保持一致，包括在质量、进度等方面的要求。在某些特殊情况下，集成测试可以结合软件配置项测试和系统测试一起完成。但是，应充分考虑到集成测试与配置项测试和系统测试的差异。一般情况下，应先完成配置项集成测试，再进行配置项测试，接着应进行系统集成测试，最后进行系统测试。另外，参与集成测试和配置项测试、系统测试的人员也不同。通常情况下，集成测试由开发人员完成，配置项测试和系统测试由相对独立的测试人员完成。

单元到配置项的集成测试与软件配置项测试之间存在较为明显的差异。它们之间的差异主要表现在以下 5 个方面。

（1）测试对象不同。配置项测试的对象是软件配置项，而配置项集成测试的对象是单元的组合，不是完整的配置项。

（2）测试的依据不同。配置项集成测试的依据是软件配置项的设计文档，而软件配置项测试的依据是软件配置项的研制任务书和（或）软件需求规格说明。

（3）测试环境不同。软件配置项集成测试可以在开发环境或仿真环境下进行，同时需要建立桩模块和驱动模块；而一般情况下配置项测试需要在目标环境下进行，并且需要开发相应的仿真程序和数据捕获程序。

（4）测试策略不同。软件配置项集成测试需要根据软件设计建立相应的测试策略，而软件配置项测试则一般采用黑盒测试。

（5）测试内容存在差异。软件配置项集成测试重点关注单元之间的接口（配置项内部接口），以及集成后的单元组合是否能够协调地完成所定义的功能。而配置项测试则需要从软件的方方面面考察软件研制任务书和（或）软件需求规格说明中描述的功能、性能、接口（配置项外部接口）、安全性、恢复性和安装性等是否符合要求。

同样，配置项到系统的集成测试与系统测试之间存在明显的差异。它们之间的差异主要表现在以下 5 个方面。

（1）测试对象不同。系统测试的对象是软件系统，而系统集成测试的对象是配置项的组合，不是完整的系统。

（2）测试的依据不同。系统集成测试的依据是系统设计，而系统测试的依据是系统的研制任务书和（或）系统需求规格说明。

（3）测试环境不同。系统集成测试可在开发环境或仿真环境下进行；而系统测试需要在目标环境下进行，并且需要开发相应的仿真程序和数据捕获程序。

（4）测试策略不同。系统集成测试需要根据系统设计建立相应的测试策略，而系统测试则一般采用黑盒测试方法。

（5）测试内容存在差异。系统集成测试重点关注配置项之间的接口（系统内部接口），以及集成后的配置项组合是否能够协调地完成所定义的功能。而系统测试则需要从系统的

方方面面考察研制任务书和(或)系统需求规格说明中描述的功能、性能、接口(配置项外部接口)、安全性、恢复性和安装性等是否符合要求。

## 5.1.3 集成测试的重要性

有的研发人员会产生这样的疑问:软件单元都完成了测试,所有问题也得到了解决,是否还有必要按照集成策略进行集成,把所有单元放在一起进行相应的配置项测试或系统测试吗?持有这种观点的研发人员不少,这样做可能带来的问题如下:

(1) 数据可能在通过不同单元时发生丢失;

(2) 单元间相互影响,其中某个单元可能导致其他单元无法正常工作;

(3) 当单元集成在一起时无法实现预期的功能;

(4) 虽然每个单元都符合性能要求,但集成后性能超出了规定的要求;

(5) 全局数据被破坏而无法使用等。

基于上述原因,有必要按照一定的集成测试策略进行集成和测试。在作者们参与的大型工程软件测试过程中,一般需要进行单元测试、部件测试、配置项测试和系统测试。随着软件工程的深入开展,逐步开展了集成测试。这样一来部分研发人员产生了困惑,不知道该如何实施集成测试。其实,部件测试也是集成测试过程中的一部分。部件测试是将部分单元一次集成为部件,然后进行的测试。因此,部件测试是集成测试过程中的一部分,完成了部分的集成测试工作。

根据开发模型的定义应首先进行软件配置项的集成测试,然后才能进行系统集成测试。系统的组成如图 5-1 所示。如图 5-1(a),系统由软件配置项(CSCI)1～$n$ 组成。如图 5-2 所示,其中某个 CSCI$i$ 由软件单元(CSU)1～$m$ 组成。有时,由于系统的复杂性,可能还存在子系统的设计,即在系统和软件配置项之间存在子系统的设计,如图 5-1(b)所示,那么集成测试还需要考虑子系统的集成测试。本文为了描述方便,选择了简单系统的设计,只描述配置项集成测试和系统集成测试。

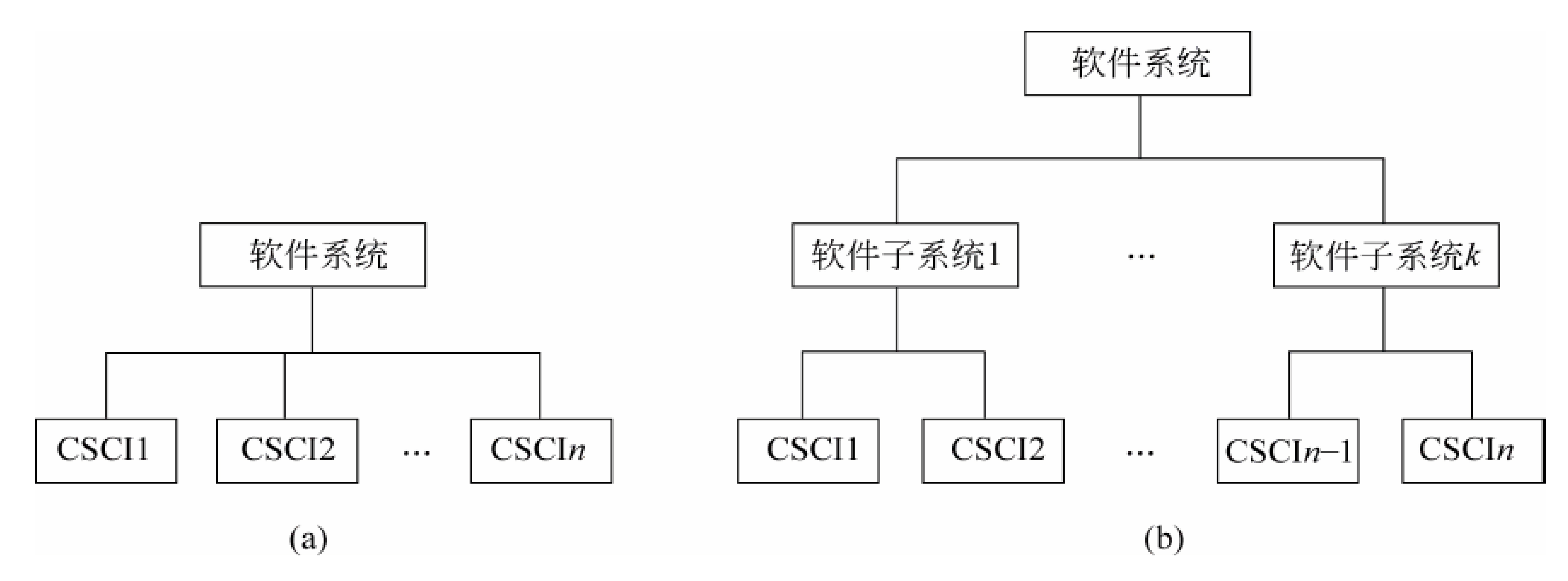



**图 5-1 系统组成**

实践证明,对于较为复杂的系统,如果不尽早地进行集成测试,等到所有软件开发完成后,只开展软件配置项测试或系统测试,常常会出现配置项或系统无法顺利集成的问题。同时,问题的定位也将会变得非常困难,甚至会出现不得不终止测试进行软件问题定位与修改

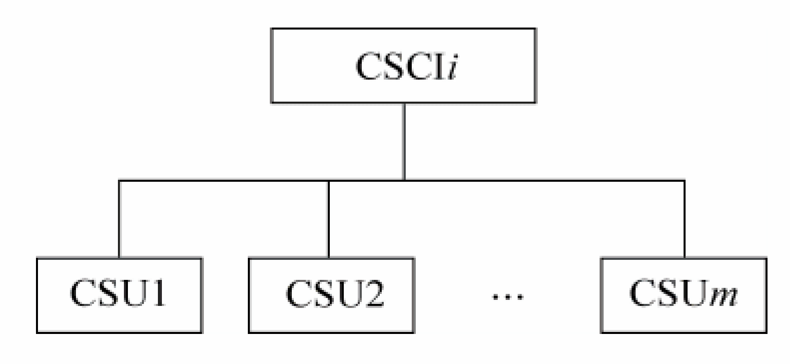


图 5-2　软件配置项结构示意图

的情况，浪费大量的时间，进度更是无法保证。另外，还会使软件中遗留不可预知的问题，为软件的正式交付运行带来隐患。

## 5.2　集成测试原则

为了有效地开展集成测试，应尽早地开始集成测试策划。在进行集成测试策划时需要遵循以下原则：

（1）在配置项集成测试前应对软件单元进行充分的测试，在系统集成前应对配置项进行充分的测试；

（2）集成测试应依据软件设计文档，配置项集成测试应依据软件概要设计说明，系统集成测试应依据系统设计说明；

（3）集成测试应分层进行，应从软件单元开始集成至软件配置项，软件配置项集成到子系统，子系统集成到系统；

（4）集成测试的策略选择应与软件开发人员充分沟通，并应当综合考虑体系结构、质量、成本和进度之间的关系，以确定最佳的集成测试策略；

（5）集成测试应覆盖所有接口；

（6）集成测试应尽早开始；

（7）当接口修改时，应对相关接口进行回归测试。

## 5.3　集成测试环境

对于某个配置项的集成测试，集成测试环境尽可能使用目标运行环境，但确实存在困难时，可分析测试环境与目标运行环境的差异，避免软/硬件环境的不一致导致的测试结果的不可信。在配置项集成测试时需要开发部分的桩模块和驱动模块，这些可能在单元测试时已具备，测试人员可以通过分析确定采用单元测试的程序和工具。

系统测试环境比较复杂，通常系统部署在不同的软/硬件平台上，系统集成测试需要对各个平台上运行的软件进行集成测试。此时，往往需要使用一些专用工具，甚至需要开发一些数据或接口仿真工具才能完成相应的测试。

集成测试环境的建立可以从以下 4 个方面考虑。

（1）硬件环境。在进行集成测试时，应尽可能地使用软件实际的运行环境。如果无法

使用实际的运行环境，可以使用有效的仿真环境进行集成测试。但是，应对仿真环境与实际运行环境的差异进行分析，确保仿真环境与实际运行环境是等效的。如果确实无法保证环境的有效性，可采用分析方法说明结果的可信性，或者将部分测试放在后续测试中执行。

（2）操作系统环境。对于大型复杂系统来说，通常会在多个操作系统平台中运行相应的软件。在进行集成测试时，需要充分考虑操作系统的要求。

（3）数据库环境。数据库环境也十分的重要，需要根据研制任务书或需求规格说明中描述的数据库环境要求建立集成测试环境。这是因为不同的数据库系统的性能、容量都存在不同，在集成测试时尤其要关注这部分内容。

（4）网络环境。网络环境是影响系统性能的关键因素，因此应按照研制要求采用实际的网络环境进行部署和配置。某系统集成测试网络环境示意图，如图 5-3 所示。

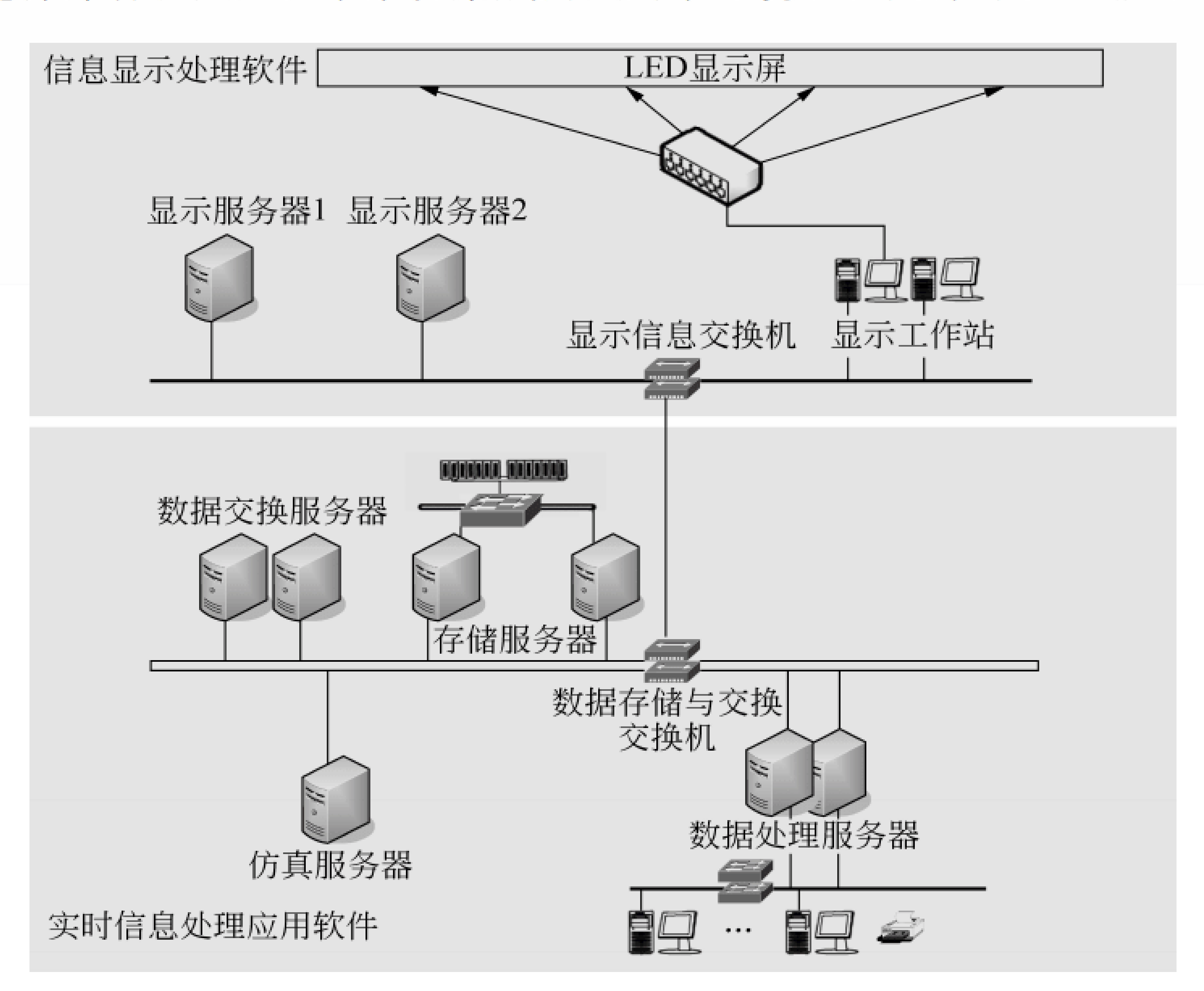



图 5-3 某系统集成测试网络环境示意图

## 5.4 集成测试策略

集成测试主要采用黑盒测试技术，并适当运用白盒测试技术。集成测试的依据是软件设计文档。

集成测试的策略有很多，例如：非增量式集成和增量式集成、三明治集成、核心系统先行集成、分层集成、基于功能的集成、高频集成、基于进度的集成、基于使用的集成、基于风险的集成和客户/服务器系统的集成等。这些集成测试策略可以分为两大类，即增量式集成和非增量式集成。非增量式集成主要是大爆炸式集成，即按软件设计将软件单元一次全部组装起来，然后进行整体测试。增量式集成包括自顶向下增量式集成、自底向上增量式集成和

三明治集成等。

两大类集成策略的优缺点主要有以下 5 个方面：

(1) 增量式集成测试需要建立的测试环境复杂，需要开发桩模块和驱动模块，工作量较大；

(2) 增量式集成测试发现的接口间问题容易定位；

(3) 增量式集成测试较为充分；

(4) 非增量式集成测试需要的测试时间少，但问题定位相对较为困难，往往影响软件研制进度；

(5) 非增量式集成测试可以进行并行测试。

在进行配置项集成测试和系统集成测试时，都需要选择适合的集成策略，因此本节介绍一些常用的集成策略，为了描述的便捷性，下面将集成测试过程中，软件单元的组合、配置项的组合统称为模块，在某些描述中独立的软件单元或软件配置项也被笼统地称作模块，有关模块的划分在 5.6.2 节中介绍。

## 5.4.1　大爆炸式集成

为了在最短的时间内完成集成，形成完整的系统，并且用最小的测试集合来验证系统的正确性，通常采用一次性集成的策略进行集成和测试。在软件研制过程中，有时会因为时间进度或系统相对较为简单等原因选择采用此策略。

### 1. 集成策略

大爆炸式集成是非增量式集成，即一次性集成或整体拼装。该方法将所有系统组成部分一次性集成完毕，不考虑系统结构以及可能存在的风险。某系统组成和集成方法示意图如图 5-4 所示。

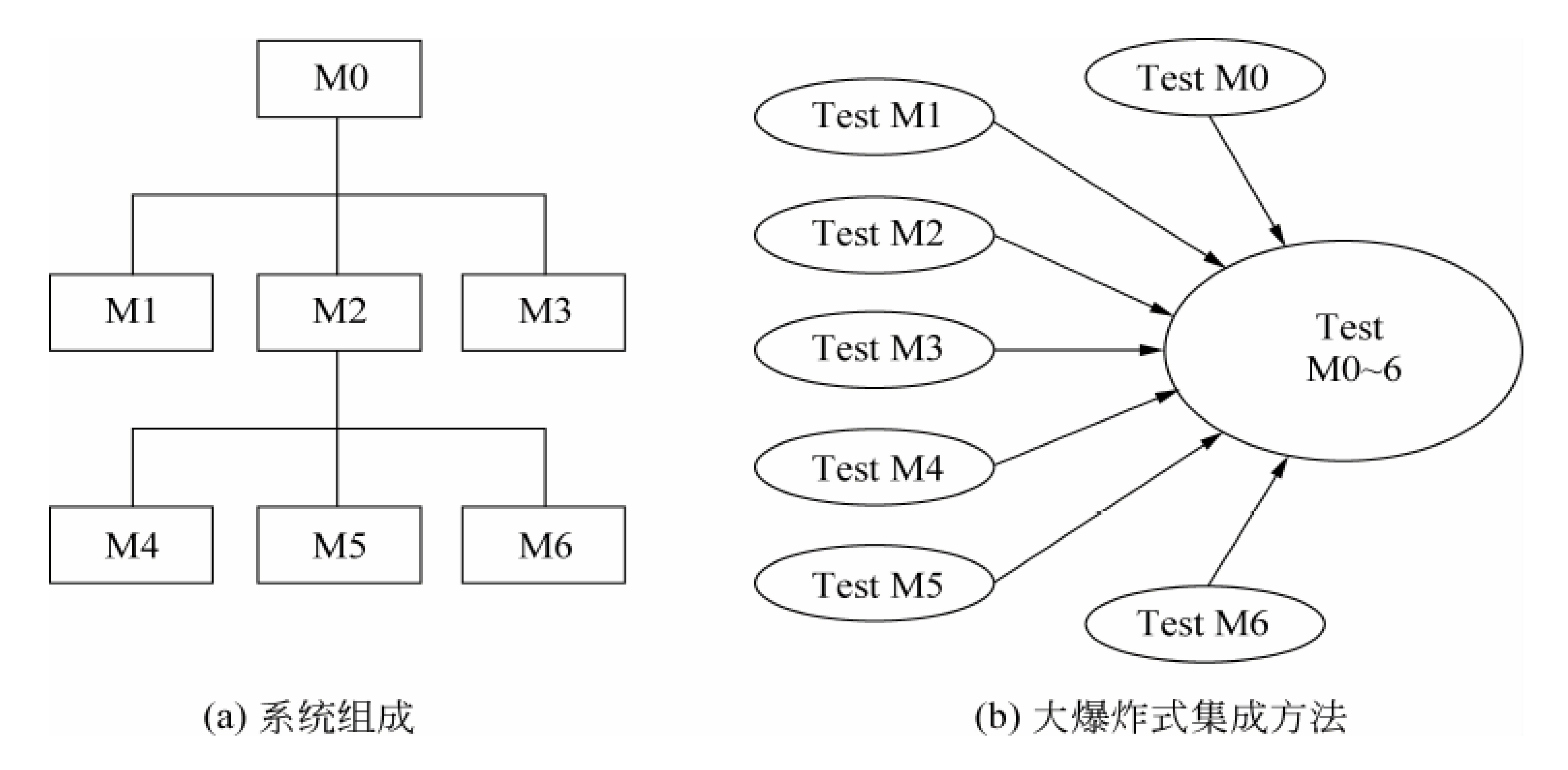



**图 5-4　大爆炸式集成方法示意图**

### 2. 优点

大爆炸式集成测试策略的优点主要有：

(1) 运用该策略可以在很短的时间内完成集成测试；

(2) 该策略需要建立的集成测试环境代价较小，只在必要的情况下开发少量的驱动模块和桩模块；

(3) 该策略不需要考虑软件设计时的结构；

(4) 集成测试的测试用例较少；

(5) 集成测试可以并行进行，需要的测试时间较少。

**3. 缺点**

大爆炸式集成测试策略的缺点主要有：

(1) 一般情况下，无法一次就集成成功；

(2) 在集成测试过程中，对发现的问题进行定位比较困难，问题定位一般耗时较长，甚至有些无法复现和定位；

(3) 容易遗留错误到系统中，造成后续测试错误较多，同时也为软件的交付运行埋下隐患。

**4. 选择原则**

选择该集成测试策略应参考如下原则：

(1) 系列工程项目中的维护过程，例如大型系列工程项目除首次研发以外的维护；

(2) 软件规模比较小，例如只有一个可执行程序的软件配置项；

(3) 软件开发团队较小，并使用集成化开发平台并行开发，并经常提交编译、调试等，例如一个规模在 2 万行左右，2 个人的开发团队，使用 VC++ 开发平台；

(4) 单元测试已通过了较为充分的测试。

## 5.4.2 自顶向下集成

自顶向下集成测试策略是依据软件设计从顶层模块开始逐步集成和测试，最后形成完整的系统。这种方法可以较早地对接口进行验证，先验证顶层模块间的接口，再逐步验证与底层模块，以及底层模块间的接口。

**1. 集成策略**

自顶向下集成是增量式集成，在集成过程中可采用深度优先策略和广度优先策略。图 5-4(a)所示系统，其深度优先策略集成方法如图 5-5 所示，其广度优先策略集成方法如图 5-6 所示，其中 S1、S2、S3、S4、S5 和 S6 为集成测试所需的桩模块。

自顶向下集成测试策略是按软件结构设计自上而下进行，即集成的顺序是首先集成主控模块，然后按照软件控制层次结构向下进行集成。从属于主控模块的模块按深度优先策略(纵向)，或者广度优先策略(横向)逐步集成。深度优先策略的集成方式是首先集成结构中的一个主控路径下的所有模块，主控路径的选择是任意的，一般根据软件的特点来确定。广度优先策略的集成方式是首先沿着水平方向，把每一层中所有直接隶属于上一层的模块集成起来，直至最底层。

自顶向下集成策略的具体步骤如下。

(1) 主模块作为驱动模块，所有与之直接相关的模块全部用桩模块代替，对主模块进行测试。若单元测试时已进行过相应测试，此步骤可以省略。

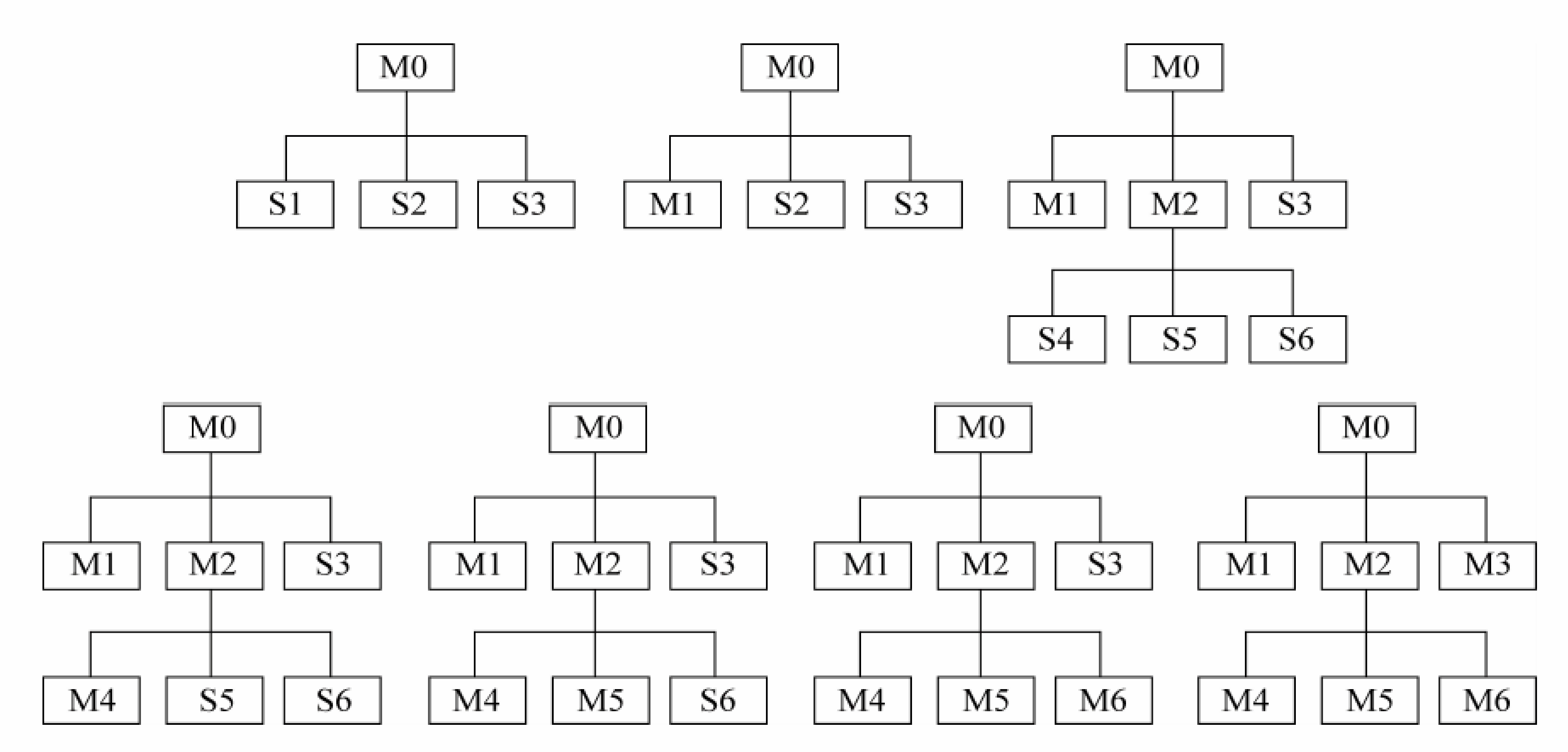



图 5-5　深度优先策略集成方法示意图

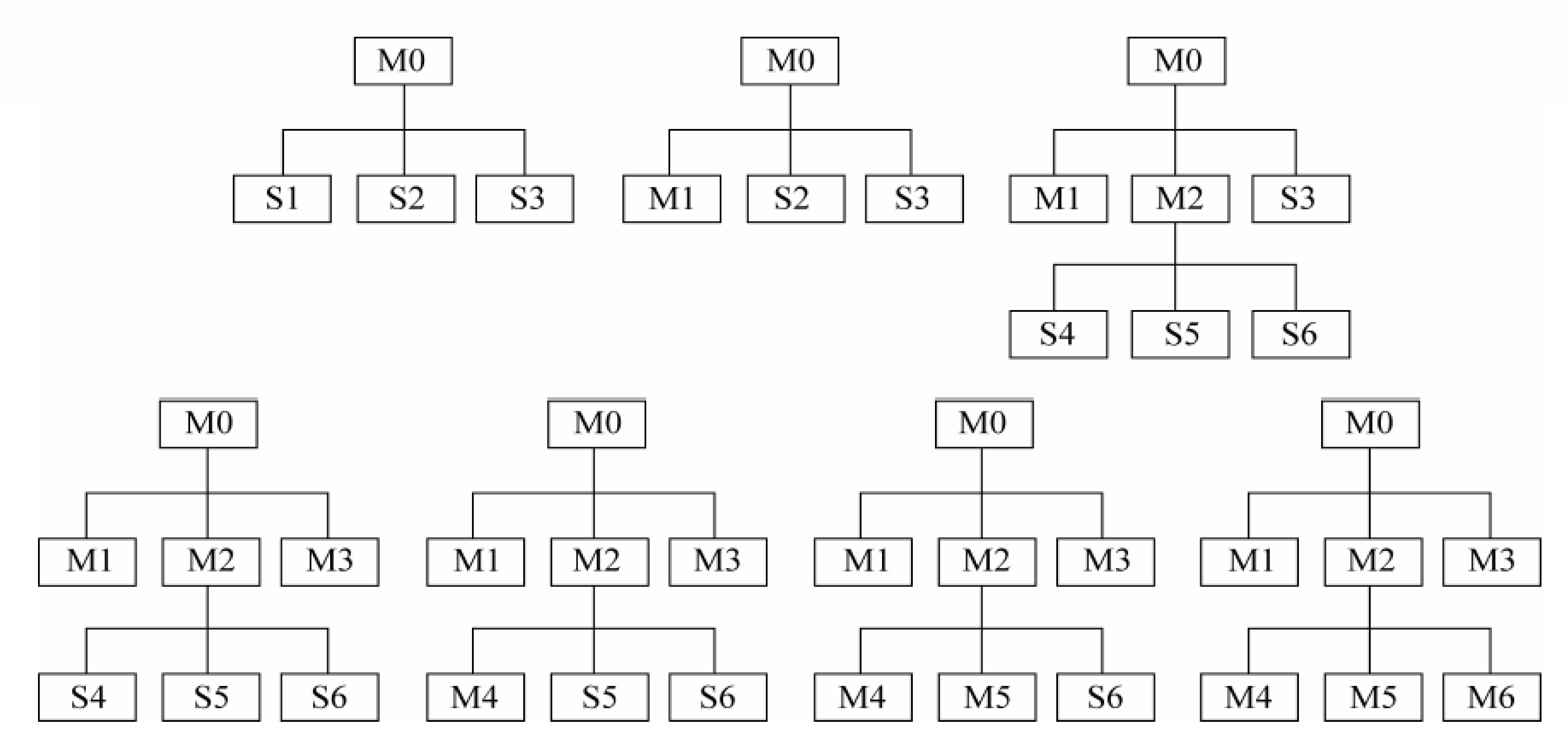



图 5-6　广度优先策略集成方法示意图

(2) 采用深度优先或广度优先策略，用实际模块替换相应桩模块，若该模块存在下属模块时，需要用桩模块替代实际模块的下属模块，集成后进行测试。

(3) 若集成过程中发现问题，需要进行相应的回归测试。

(4) 重复(2)和(3)直至所有模块都集成完毕。

**2. 优点**

自顶向下集成测试策略的优点主要有：

(1) 由于关键的控制流程在较高层次模块中，因此采用此策略进行集成测试可较早地开展关键控制流程的测试，以便较早地发现关键控制流程中存在的问题，并尽早修复关键的控制流程错误；

(2) 选用深度优先集成策略，可以验证完整的软件功能，可以先完成逻辑输入分支的测试，为后续主要处理流程的测试提供保证；

(3) 软件功能可较早地得到验证，为用户提供对软件的直观了解；

(4) 不需要开发驱动模块；

(5) 由于这种方式与设计顺序一致,与设计的符合性较好;
(6) 可以与设计、编码、单元测试等并行开展;
(7) 缺陷定位较为容易,例如由于某模块加入后,测试出现问题,那么可以确定新加入的模块存在问题,或者新加入的模块与已集成的模块之间的接口存在问题。

**3. 缺点**

自顶向下集成测试策略的缺点:
(1) 建立集成测试环境时需要开发大量的桩模块,工作量较大;
(2) 高层模块需要较好的可测试性;
(3) 对底层模块的验证较晚,导致底层模块的测试充分性难以保证。

**4. 选择原则**

该方法适合于结构简单、结构化编程方法的软件产品。选择该集成测试策略应参考如下原则:
(1) 软件控制接口清晰;
(2) 高层接口变化较小;
(3) 底层接口未定义或可能发生变化;
(4) 控制模块存在较大技术风险,需要尽早开展验证;
(5) 用户希望尽早了解系统功能。

### 5.4.3 自底向上集成

自底向上集成测试策略是从软件的最底层模块开始,按照结构关系,逐层向上集成,以此验证系统。

**1. 集成策略**

自底向上集成是增量式集成,自底向上集成策略是从软件结构的最底层模块开始集成和测试,按照软件结构逐步向上直至顶层模块。采用该策略进行集成测试,对于某个模块来说,其下属模块已完成集成和测试。所以,在该模块的测试时不需要再开发桩模块。

自底向上集成策略的具体步骤如下:
(1) 对软件设计结构中的最底层的模块进行测试,此时可以将多个最底层模块并行进行测试,或者把只有一个最底层模块的父模块与其组合在一起进行测试;
(2) 使用驱动模块对上述步骤中确定的模块(或模块组合)进行测试;
(3) 将完成测试的模块的直接上层模块集成进来,使用相应的驱动模块进行测试;
(4) 重复上述步骤,直到将系统的最顶层模块集成进来,并完成测试。

图 5-4(a)所示的系统,其自底向上集成方法如图 5-7 所示。其中,D1、D2、D3、D4、D5 和 D6 为集成测试所需的驱动模块。

**2. 优点**

自底向上集成测试策略的优点主要有:
(1) 对底层模块的验证可以较早地开展;
(2) 对底层模块的验证可以并行进行,可以有效地提高软件开发的效率;

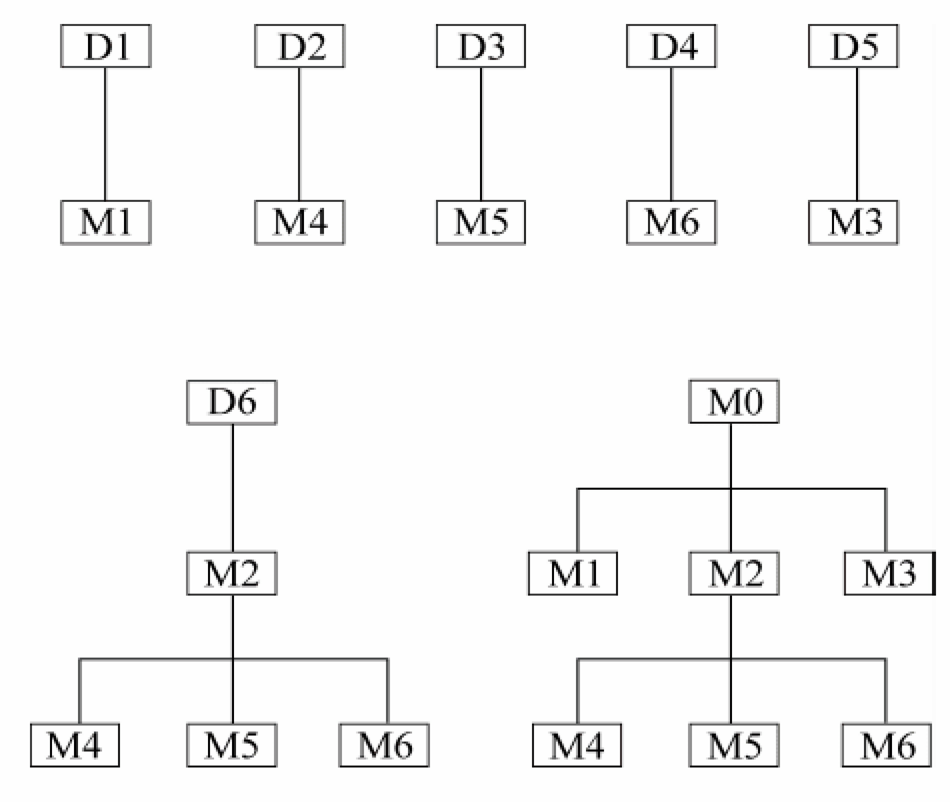



图 5-7　自底向上集成方法示意图

(3) 测试中所需的驱动模块是针对测试进行开发的，可以有效地保证测试的可测试性和有效性；

(4) 减少了桩模块的开发工作量，只需要在一些特殊测试要求时开发少量的桩模块；

(5) 缺陷定位也较为容易。

#### 3. 缺点

自底向上集成测试策略的缺点主要有：

(1) 需要开发驱动模块；

(2) 对高层、顶层模块的验证较晚，某些设计上的问题不能尽早发现；

(3) 集成到后期顶层时，此时系统非常复杂，对底层异常的覆盖非常困难。

#### 4. 选择原则

该方法适合于结构化编程方法的软件产品。选择此方法集成测试时应参考如下原则：

(1) 软件结构比较简单；

(2) 底层接口较为稳定；

(3) 高层接口变化较频繁；

(4) 底层模块能够比较早地完成开发。

### 5.4.4　三明治式集成

三明治式集成综合了自顶向下和自底向上集成策略的优点，将系统划分为 3 层，中间层为目标层。集成测试时，对 3 层分别进行集成和测试，最后再将 3 层集成到一起进行测试。

#### 1. 集成策略

三明治式集成将系统划分为 3 层，对上层采用自顶向下的集成策略进行集成测试，对下层采用自底向上的集成策略进行集成测试，对中间层进行独立的测试，将中间层与下层集成和测试，最后将 3 层一起集成并进行测试。

三明治式集成策略的具体步骤如下：

(1) 对上层进行自顶向下的集成测试，需要桩模块的辅助；

（2）对下层进行自底向上的集成测试，需要驱动模块的辅助；

（3）对中间层和下层进行集成测试，需要驱动模块的辅助；

（4）将三层集成到一起进行测试。

图 5-4(a)所示的系统，将其分成上层、中间层和下层，三明治式集成测试策略如图 5-8 所示。其中，S1、S2、S3 为上层测试所需桩模块，D2、D3、D4 为下层测试所需的驱动模块，S4、S5、S6 、D1、D5 和 D6 为中间层测试所需桩模块和驱动模块。

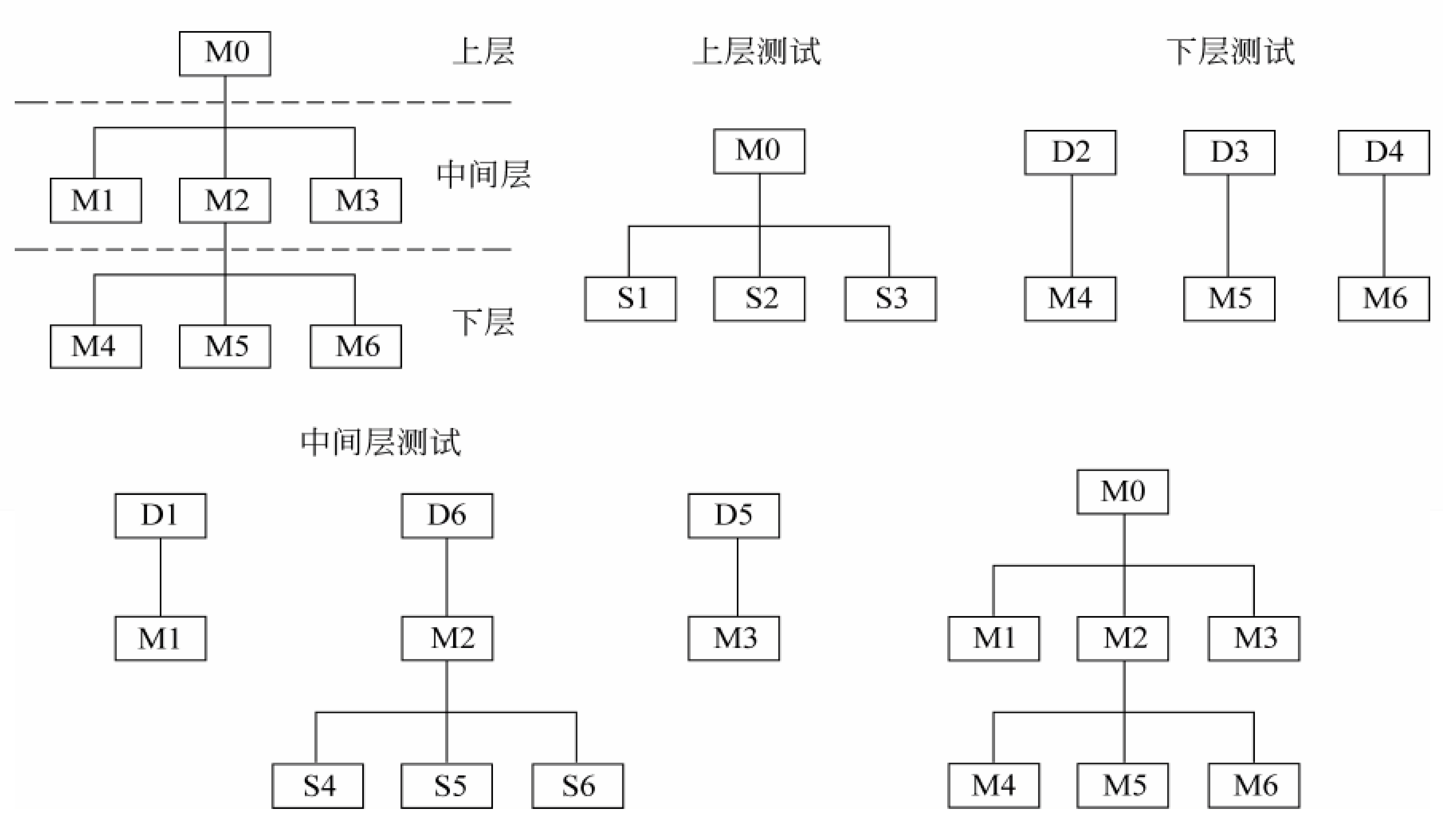



**图 5-8　三明治式集成方法示意图**

**2. 优点**

三明治式集成测试策略的优点主要有：

（1）具有自顶向下和自底向上集成测试策略的优点；

（2）能够并行地进行上层、中间层和下层的测试，有效保证测试进度。

**3. 缺点**

三明治式集成测试策略的缺点是中间层选取比较困难。如果选择不恰当，将造成开发驱动模块和桩模块的工作量较大。

**4. 选择原则**

大部分软件开发项目都适用该策略。

## 5.4.5　核心系统先行集成

核心系统先行集成策略结合了自顶向下、自底向上和大爆炸集成测试策略，此策略用来验证紧耦合系统。

**1. 集成策略**

核心系统先行集成测试策略的思想是先对内核部分进行集成测试，在测试通过的基础

上再按各外围模块的重要程度逐个进行集成。

核心集成测试步骤如下。

(1) 对核心模块进行充分的测试。

(2) 将核心模块使用大爆炸式集成策略进行一次性集成，并进行测试，解决集成中出现的各类问题。在核心模块相对较多的情况下，也可以按照自底向上的步骤进行核心模块的集成。

(3) 按照各外围模块的重要程度以及模块间的相互制约关系，确定外围模块集成到核心系统中的顺序。

(4) 在外围模块集成到核心系统之前，应对步骤(3)中确定的外围模块完成集成测试。

(5) 按步骤(3)确定的顺序加入外围模块，形成最终的系统。

**2. 优点**

核心系统先行集成测试策略的优点是对于快速软件开发很有效，是较复杂系统的集成测试方法，能有效保证重要功能和服务的实现。

**3. 缺点**

核心系统先行集成测试策略的缺点主要有：

(1) 划分核心模块和外围模块较为困难；

(2) 集成测试需要开发较多的桩模块和驱动模块；

(3) 由于对核心部分采用大爆炸式集成策略，因此存在接口测试不完整的可能性。

**4. 选择原则**

该策略适合大型复杂系统，适用该策略的情况如下：

(1) 具有多层协议的嵌入式系统；

(2) 操作系统。

## 5.4.6 分层集成

分层集成测试策略是将具有层次结构的系统，先按照其层次结构分别选择恰当的集成测试策略，然后再按照选定的集成测试策略完成所有层次的集成测试。

**1. 集成策略**

分层集成就是通过增量方法集成测试一个具有层次体系结构的应用系统。分层模型常用于通信系统的集成测试。这是因为通信系统在设计时就是按照分层设计的。图 5-9 所示为某系统组成和集成方法示意图。

分层集成测试的步骤如下。

(1) 按照系统的分层设计，确定每个层次内部的集成策略，并分别进行集成测试。层内的集成策略可以采用自顶向下、自底向上等方法，甚至可以采用非增量集成(例如大爆炸集成)等；

(2) 确定层间的集成策略，可以采用自顶向下、自底向上等方法，甚至可以采用非增量集成等策略，并按照层间的集成策略进行集成测试。

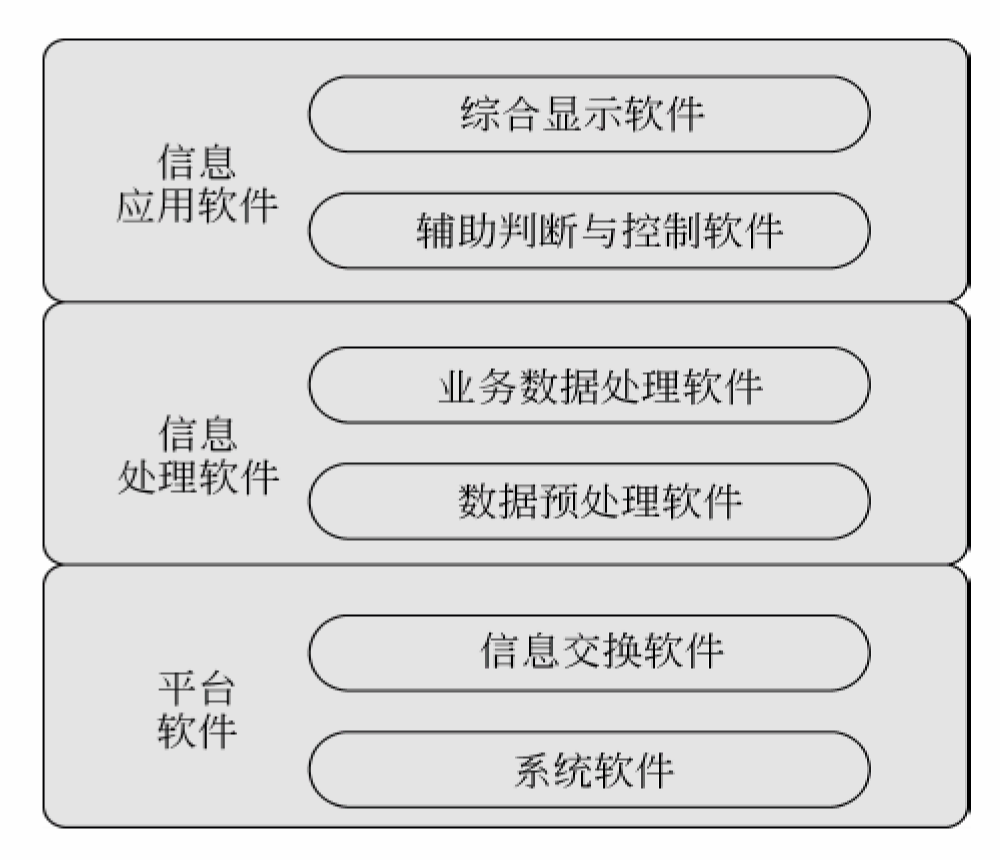



图 5-9 分层集成方法示意图

**2. 优点**

分层集成测试策略的优点与其使用的各层次集成测试策略和层间集成测试策略的优点一致。

**3. 缺点**

分层集成测试策略的缺点与其使用的各层次集成测试策略和层间集成测试策略的缺点一致。

**4. 选择原则**

该方法适用于层次结构清晰的系统。

## 5.4.7 基于功能的集成

基于功能的集成属于增量式集成方法,使用该策略进行集成测试可以尽早地开展系统关键功能的验证。

**1. 集成策略**

在系统的开发过程中,用户和开发人员都希望尽早地看到软件主要功能的实现与展示,这对提升用户和开发团队的信心是非常有益的。另外,还可以使用户尽早地确认系统的主要功能,避免需求理解的问题。基于功能的集成策略是从系统功能的优先级出发,对集成顺序进行定义,进而实现集成和测试的方法。

基于功能的集成测试的步骤如下。

(1) 按照一定的准则确定各功能的优先级。

(2) 分析与优先级最高的功能相关的功能,把这些功能依次集成起来进行测试。

(3) 选择在步骤(2)中未集成的下一个关键功能,分析与其相关的功能,把它们依次集成起来进行测试。

(4) 重复步骤(3),直至所有模块都完成集成和测试。

#### 2. 优点

基于功能的集成测试策略的优点主要有：

（1）可以尽早地验证关键功能的正确性；

（2）由于集成某些关键功能时，需要同时集成与其相关的模块，因此比自顶向下和自底向上集成策略要快捷；

（3）接口测试的覆盖率较自顶向下和自底向上集成测试策略要低；

（4）驱动模块的开发工作量较少。

#### 3. 缺点

基于功能的集成测试策略的缺点主要有：

（1）对于复杂系统，很难分析清楚功能之间的相互关系；

（2）对相关功能之间的接口测试不充分；

（3）初始测试需要使用较多的桩模块；

（4）冗余测试较多。

#### 4. 选择原则

选择该集成测试策略应参考如下原则：

（1）系统的主要功能存在较大风险，存在关键技术需要突破；

（2）系统的主要功能需要尽早参与联调或联试。

### 5.4.8　高频集成

目前，越来越多的系统采用迭代式开发或增量式开发模型，采用此模型的软件项目需要对每一次增量后所得到的系统进行集成和测试，以免最终获得的系统与用户要求出现较大偏差。

#### 1. 集成策略

高频集成是增量式集成。第一次是一个基本功能集成包，随着下一个增量开发的完成，需要集成和测试增大后的系统是否能够稳定工作，功能是否正确。如果在整个系统完成后再实施集成测试，其集成和测试的难度将非常大。

高频集成测试的步骤如下。

（1）开发人员和测试人员同步进行代码开发、测试用例设计、测试软件开发等。

（2）对新增的代码进行测试。

（3）将新增代码集成到已完成的集成包中。

（4）针对集成后的测试包进行集成测试。

#### 2. 优点

高频集成测试策略的优点主要有：

（1）开发和测试同步进行，具有同样的重要性；

（2）能够较早地发现严重错误，可避免向下传递；

（3）桩模块不是必需的；

（4）每次集成测试后的系统都可以向用户交付，通过不断地向用户交付一定的功能，使开发人员、用户能通过交付的系统及时地进行沟通，以便使系统能够逐步地向完整系统接近。

**3. 缺点**

高频集成测试策略的缺点主要有：

（1）根据增量划分的情况，测试有时可能过于简单，因此存在无法发现深层次问题的情况；

（2）在开发、集成的初期可能无法有条不紊地集成。

**4. 选择原则**

该方法适合于采用迭代或增量开发模型的软件项目。使用高频集成方法需要具备以下条件：

（1）新的增量应较为稳定，并且已完成的增量应已通过验证；

（2）软件编码和测试工作可以并行进行；

（3）需要使用一定的自动化工具进行测试；

（4）软件版本必须得到有效控制，一般情况下应使用工具完成。

## 5.4.9 基于进度的集成

进度是几乎每个软件开发项目都会面对的压力。为了按进度完成开发任务，往往忽略了部分的质量要求。基于进度的集成就是试图在兼顾质量和进度两者之间达到一种平衡。

**1. 集成策略**

这种集成方法就是把先完成的单元先集成，并根据情况开发驱动模块和桩模块，使集成测试尽可能的与开发进度保持一致，进而缩短项目研制周期。

**2. 优点**

该方法的优点在于开发工作和集成测试工作可以并行进行，较为有效地缩短了项目周期。

**3. 缺点**

基于进度的集成测试策略的缺点主要有：

（1）最先得到的模块可能相互之间不存在关联性，只能进行模块的测试，导致许多接口需要等到后期进行测试，此时系统已经变得较为复杂，无法有效地发现系统存在的问题；

（2）桩模块和驱动模块的开发工作量可能会比较大。

**4. 选择原则**

进度要求高于质量要求的项目可选用此策略进行集成测试。

## 5.4.10 基于使用的集成

基于使用的集成测试策略主要用于面向对象方法开发的系统，可以通过类之间的使用

关系来进行集成测试。

**1. 集成策略**

基于使用的集成测试策略，通过从独立的类开始集成，逐步扩大到有依赖关系的类，最后集成为完整的系统。通过这样的方法可测试类之间接口的正确性。这种方法的步骤如下。

(1) 分析类之间的关联关系。

(2) 根据类之间的耦合关系，首先测试独立的类。

(3) 测试服务器类。

(4) 逐步增加具有依赖性的类，直到整个系统被集成完毕。

**2. 优点**

基于使用的集成测试策略的优点与自底向上测试策略相似。

**3. 缺点**

基于使用的集成测试策略的缺点与自底向上测试策略相似。

**4. 选择原则**

使用面向对象方法开发的系统集成适用该策略。

### 5.4.11　基于风险的集成

在软件开发过程中为了尽快地对高风险模块进行验证，可采用基于风险的集成测试策略。

**1. 集成策略**

基于风险的集成是为了避免系统中某些存在高风险的模块对系统造成危害，需要尽早地开展这些模块的验证，保证系统的稳定性。该方法类似基于功能的集成。

**2. 优点**

基于风险的集成测试策略的优点是风险最高的模块优先进行验证，以便保证整个系统的有序集成。

**3. 缺点**

基于风险的集成测试策略的缺点是需要对系统中各模块的风险有清晰、准确地分析和判断。

**4. 选择原则**

基于风险的集成测试策略适用于部分模块存在较大的风险，这些风险可能会影响到进度、对外接口等的系统。

### 5.4.12　客户/服务器系统的集成

目前，许多软件系统是基于客户/服务器架构实现的，因此需要针对此类软件设计客户/

服务器系统的集成测试策略。

**1. 集成策略**

在基于客户/服务器架构开发的系统中，服务器对客户的消息做出响应，客户对来自系统的消息做出反应。

基于客户/服务器系统的集成测试的步骤如下。

(1) 单独测试客户端和服务器端。

(2) 把第一个客户端与服务器端进行集成和测试。

(3) 把下一个客户端与步骤(2)完成的系统进行集成和测试。

(4) 重复步骤(3)，直到所有客户端都已完成集成和测试。

**2. 优点**

基于客户/服务器系统的集成测试策略的优点主要有：

(1) 避免了大爆炸集成的风险；

(2) 集成次序没有严格的约束，可以结合功能优先和风险优先策略进行集成。

**3. 缺点**

基于客户/服务器系统的集成测试策略的缺点是需要较大量的桩模块和驱动模块的开发。

**4. 选择原则**

该策略适合基于客户/服务器架构设计的系统。

## 5.5 集成测试内容

集成测试与其他测试级别不同，集成测试重点关注软件模块之间是否能够协调一致的工作，测试时需要重点关注的内容包括：

(1) 在集成时模块的数据是否丢失；

(2) 模块集成后是否能够协调地完成定义的组合功能；

(3) 一个模块的功能是否对其他模块的功能产生影响；

(4) 全局数据是否存在被异常修改的问题；

(5) 单个模块的误差是否会被积累而超出预期的要求。

集成测试常见问题如下：

(1) 对需集成的模块的软件版本控制存在问题；

(2) 存在遗漏、重叠或冲突的函数；

(3) 文件或数据库使用不正确或不一致的数据结构；

(4) 文件或数据库使用冲突的数据视图/用法；

(5) 破坏全局数据的完整性；

(6) 客户发送违反服务器前提条件、顺序约束的消息；

(7) 错误参数或不正确的参数值；

(8) 进程和(或)线程之间冲突；

（9）资源冲突。

## 5.6　测试方法

集成测试一般情况下采用动态测试的黑盒测试方法，需要根据软件体系结构设计和软件的特点、质量进度要求等确定集成测试策略。

为了有效地开展集成测试，必须依据软件设计文档做好体系结构分析、模块分析、接口分析、可测试性分析和集成测试策略分析。

### 5.6.1　体系结构分析

体系结构分析是进行集成测试的基础，集成测试人员需要根据体系结构分析的结果来确定恰当的集成测试策略。体系结构的分析应从以下两个方面进行：

（1）从需求出发，划分出系统或软件配置项需求的结构层次，这个结构层次可以帮助确定集成时的层次；

（2）依据软件设计文档确定模块之间的相互关系，以便确定集成测试的粒度。

### 5.6.2　模块分析

模块分析是集成测试最重要的工作之一。模块划分的好坏直接影响集成测试的工作量、质量和进度。

一般模块划分应从以下 4 个方面进行考虑：

（1）确定关键模块，这里的关键模块主要是指结构关键、功能关键或安全关键模块等；

（2）与某一模块相关的其他模块，可按一定的顺序进行排序；

（3）在制定集成顺序时，应将模块间关系最为紧密的模块先进行集成；

（4）分析模块间的信息交互是否容易被模拟。

集成测试模块划分的原则如下：

（1）模块间关系紧密的可作为一个模块进行集成；

（2）与外围模块间的关系简单、松散；

（3）需要建立的桩模块和驱动模块简单，便于构造。

为了提高集成测试的效率，应根据一定的原则识别关键模块。关键模块的特征主要包括如下 5 个方面：

（1）和多个软件需求有关，或与关键功能相关；

（2）处于控制结构的顶层；

（3）模块复杂或容易出现错误；

（4）含有关键的性能需求；

（5）被频繁使用的模块。

关键模块分析的方法如下：

(1) 与开发人员交流,获得关键模块的信息;

(2) 通过静态分析工具来进行分析,寻找高内聚、频繁使用的模块,或处于控制顶层的模块;

(3) 依据需求跟踪矩阵来识别关键需求所对应的关键模块;

(4) 对于维护性项目可以根据历史数据进行分析,包括:变更的影响、缺陷高密度处等;

(5) 对于新研项目可以通过文档审查、代码审查和代码走查来确定高风险模块。

### 5.6.3 接口分析

集成测试关注的重点是接口的正确性、安全性、完整性和稳定性等。因此,应对模块间的接口进行详细地分析,以便保证接口测试的充分性。分析的内容应包括:接口划分、接口分类和通过接口的数据分析等。

**1. 接口的划分**

接口的划分应根据软件设计文档开展,具体的步骤如下:

(1) 根据模块划分的结果确定模块的内部接口和外部接口;

(2) 确定软件配置项的外部接口(包括与硬件的接口);

(3) 确定子系统的外部接口(包括与硬件的接口);

(4) 确定系统的外部接口(包括与硬件、操作系统和第三方软件的接口)。

**2. 接口分类**

接口的类型可以分为两大类:

(1) 系统内部接口,指系统内部各模块之间的接口,这类接口是集成测试的重点;

(2) 系统外部接口,指系统与外部系统的接口,包括人、硬件和软件的接口或操作界面,这类接口一般在系统测试时完成。

对于内部接口可分为:

(1) 函数接口,模块间通过函数的调用来进行交互;

(2) 消息接口,消息接口常用在面向对象系统和嵌入式系统,使用这种接口时,软件模块间不直接发生联系,而是通过消息进行交互;

(3) 类接口,类接口用在面向对象系统,类接口一般通过继承、参数类、不同类方法调用等方式实现交互;

(4) 其他接口,包括全局变量、配置文件、中断、数据库等。

**3. 接口数据分析**

接口数据分析就是要分析通过接口的数据。通过对接口数据的分析可设计相应的测试用例。对不同的接口类型,其分析方法也存在差异,如:

(1) 函数接口,对于函数接口需要关注函数的参数个数、参数类型、参数顺序、参数等价类、参数的边界等情况;

(2) 消息接口,对于消息接口需要分析消息的类型、消息的域、域的属性、域的取值范围、可能的异常值等情况;

(3) 类接口，对于类接口需要分析类的属性，应重点关注公共属性和保护属性，必要时需要对私有属性进行分析，应关注属性的等价类和边界值；

(4) 其他接口，对于其他接口要分析其读写属性、并发性、等价类和边界值，对于配置文件这类接口还需要重点关注数据变化组合的情况。

### 5.6.4 可测试性分析

随着软件工程技术的发展，可测试性需求在软件需求分析阶段已作为软件的需求被分析和提出，并随着项目的进展被设计和实现。尽早地进行可测试性分析对集成测试以及后续的系统测试都是非常必要的。

对于集成测试的可测试性分析重点在于接口方面的可测试性分析。若在早期不进行可测试性分析，可能导致需要开发大量的接口测试程序。

### 5.6.5 集成测试策略分析

集成测试策略分析是需要根据被测软件的具体情况选择合适的集成策略。在 5.4 节中介绍了很多集成测试策略，在实际应用时，需要根据体系结构分析、模块分析、接口分析、可测试性分析、风险分析、人力资源分析、测试环境分析以及进度要求综合选择测试策略。在实际的软件研制项目中，一般采用多种策略相结合的方式进行集成测试。

确定集成测试策略时需要从以下两个方面考虑：

(1) 能够实现较为充分的测试，特别是关键模块；

(2) 能够将模块与接口进行清晰地划分，能够使集成测试的工作量尽可能少。

## 5.7 集成测试用例设计

集成测试是根据系统的体系结构，通过集成测试分析，确定集成测试的策略。集成测试用例设计需要针对集成测试的目的，设计有效的测试用例，以便保证通过集成测试后的系统能够协调一致地进行工作。

集成测试用例设计可以从以下两个方面考虑。

(1) 设计正常测试用例。集成测试较为关键的内容是验证软件能够协调一致地完成各项功能、性能等。因此，测试用例设计时，应依据软件概要设计中描述的主要功能和接口要求，使用等价类方法设计测试用例。同时，应根据输入域、输出域、状态转换设计有针对性的测试用例。

(2) 设计异常测试用例。根据集成测试关注的内容，异常测试通常包括：接口是否存在遗漏、接口格式是否存在错误和对接口数据异常是否能够进行必要的处理等。常用的测试用例设计方法包括：猜错法、基于风险的测试、基于故障的测试、边界测试、特殊值测试、状态转换测试。

集成测试是整个软件研制过程中的一个环节，是所有测试活动的一部分，需要根据成

本、进度和质量 3 个因素综合考虑。集成测试需要突出其特点，通常情况下应覆盖所有的接口。

## 5.8 集成测试过程

集成测试依据软件设计文档，按照如图 5-10 所示的过程开展测试，集成测试过程分为 4 个阶段。

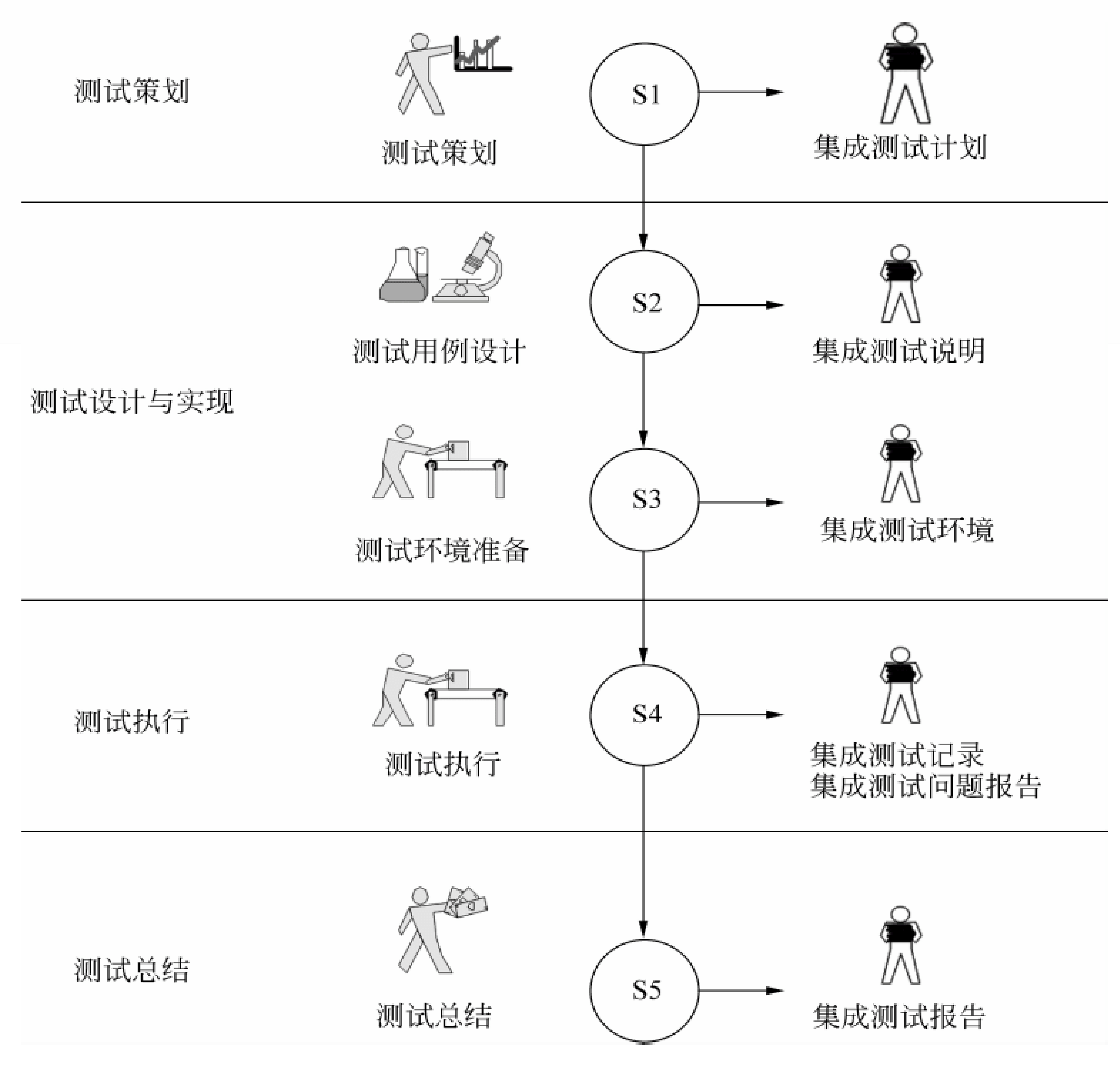



**图 5-10 集成测试过程**

(1) 集成测试策划。依据软件设计文档进行集成测试策划，制定集成测试计划。

(2) 集成测试设计与实现。依据软件设计文档、集成测试计划设计集成测试用例，开发必要的集成测试桩模块和驱动模块。

(3) 集成测试执行。按照集成测试计划、测试说明中明确的测试策略、测试环境，逐步集成并执行相应的集成测试用例，记录实测结果和测试过程出现的问题。当软件问题修改后还需要进行回归测试。

(4) 集成测试总结。根据实测结果、期望结果和评估准则分析测试数据，形成测试报告。测试报告完成后，应对整个集成测试的情况、文档等进行评审，以确保集成测试有效执行。

### 5.8.1 测试策划

为了保证软件研制进度，一般情况下可在软件设计完成后即开展软件集成测试策划、测试设计与实现工作，完成集成测试策划工作时应编写软件集成测试计划。

目前，软件研制项目在集成测试方面还存在着许多问题。主要表现在以下 3 个方面。

（1）对集成测试的重要性认识不足。

（2）集成测试开始较晚。大部分由于资源限制或集成测试策略不恰当，在整个项目编码和单元测试结束后才开始进行集成测试。而没有根据软件具体情况，尽早开展集成测试策划，选择适合的集成测试策略，分步骤、分阶段地进行集成测试。

（3）集成测试用例设计不够全面。大部分问题主要表现在，只关注了集成后的软件功能，未充分覆盖软件接口。另外，很少考虑异常情况的测试。

为了实现质量、进度、效率的平衡，应在集成测试策划时考虑如下因素：

（1）软件的质量要求；

（2）软件研制的进度安排；

（3）软件模块的优先级；

（4）软件接口的优先级；

（5）测试资源的限制。

集成测试策划的内容包括：

（1）按照软件设计和软件质量要求确定软件集成测试的要求。包括：

① 确定需要进行集成的软件单元或配置项的名称、标识；

② 明确集成测试策略，包括采取的集成策略和集成顺序；

③ 明确每一次集成后需要测试的内容；

④ 提出每次集成后的测试方法；

⑤ 提出集成测试的充分性要求；

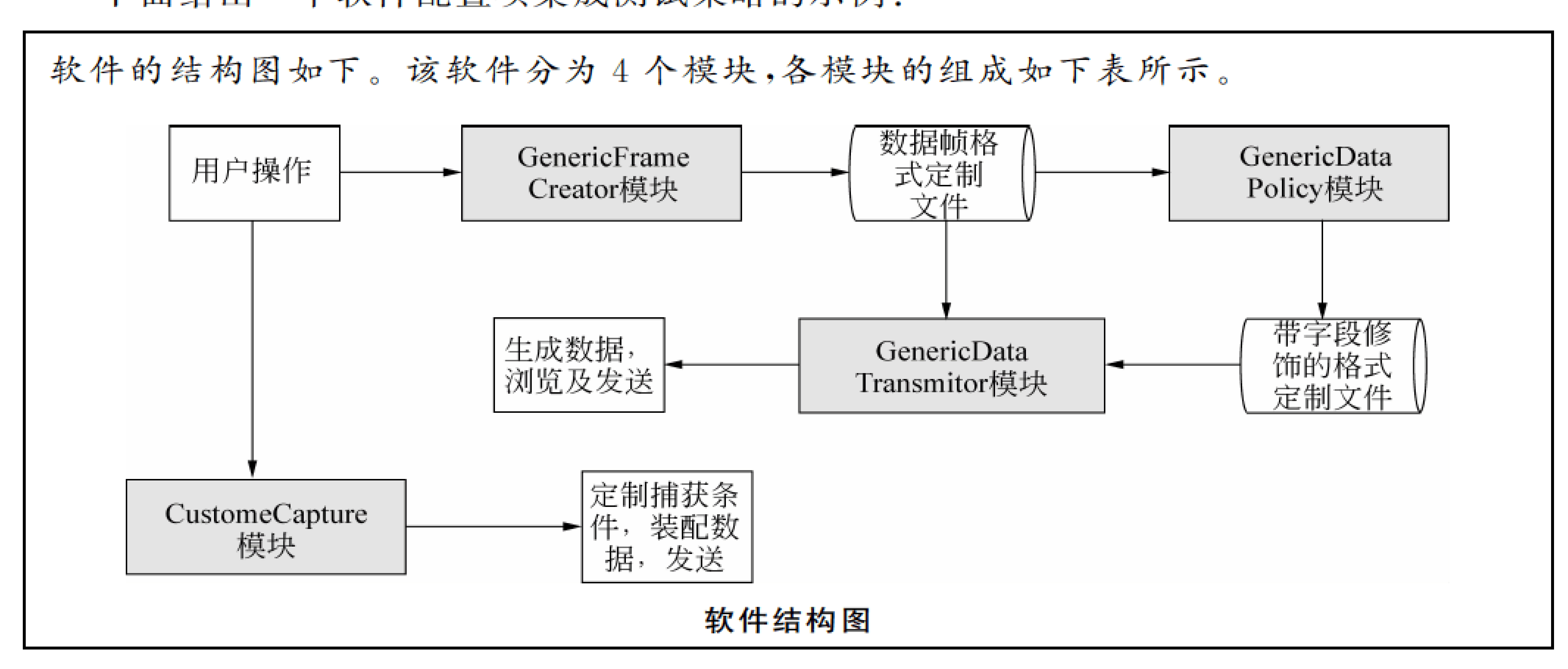
⑥ 测试终止条件；

⑦ 优先级；

⑧ 对软件设计文档的追踪关系。

下面给出一个软件配置项集成测试策略的示例：

软件的结构图如下。该软件分为 4 个模块，各模块的组成如下表所示。




软件结构图

| 序号 | 模块名称及标识 | 所包含的类的标识 |
|---|---|---|
| 1 | Generic Frame Creator | FrameStruct、DataField、IDataPolicy、TimeInDay、BcdTime、JsData、GradualData<T>、ByteGradual、UshortGradual、ShortGradual、UintGradual、IntGradual、FloatDividual、DoubleDividual、FixedData、JsData、RandomData、DataFieldFactory、DataPolicyFactory |
| 2 | Generic Data-Transmitor | eEndian、FieldCategoryAttribute、NodeElementName、IDataGenerator、BinaryFile、LineDataFile、FixedData、RandomData、RandomDataFixedLength、IInputData、InputTime、InputFixed、InputInteger、InputDecimal、InputDate、InputRandom |
| 3 | Generic Data Policy | eDecoratePosition、DecoraterFactory、FieldDecorater、UbyteDecorator、SbyteDecorator、UshortDecorator、ShortDecorator、UintDecorator、IntDecorator、UlongDecorator、LongDecorator |
| 4 | Custom Capture | INetServicer、TcpServer、UdpMulticastReceiver、UdpMulticastSender、TcpClientProc、IpPacket、TcpPacket、UdpPacket、IpProtocol、IpVersion、RawSocket、ICapturer、RawIpCapturer、IPacketParsor、RawIpPacketParsor、IPacketFilter、MultiMediaTimer、HiPerfCounter |

进行集成测试时，采用自底向上的集成顺序，具体策略如下。

① 将4个模块独立集成并进行测试，集成策略采用大爆炸方式。

② 考虑模块间的相关性和测试的便捷性。首先，将模块1与模块2进行集成。具体方法是由模块1生成自定义数据帧格式文件，然后由模块2读取数据帧格式定义文件，生成数据后发送。

③ 集成模块3。由模块1生成自定义数据帧格式文件，接着由模块3读取数据帧格式定义文件，对数据帧中定义中的每个字段进行前缀和后缀的修饰，生成带字段修饰的文件，最后由模块2读取并生成数据发送。

④ 集成模块4并进行测试。

在测试用例和测试数据设计方面，采用等价类划分、边界值分析、猜错等方法，设计包括正常值、异常值、边界值等在内的测试用例和测试数据，达到对软件功能、性能、接口的全覆盖。

根据被测软件的特点，被测软件在仿真环境中运行，测试时通过网络输入正常测试数据和异常测试数据，同时捕获输出结果，并进行分析。

（2）提出集成测试环境要求，包括测试所需的桩模块和驱动模块等。

（3）提出集成测试人员安排。

（4）安排集成测试的进度计划。应依据测试需求、测试环境、测试人员等情况，制定合理可行的软件集成测试进度计划。

（5）制定集成测试通过的准则。集成测试通过的准则示例如下：

① 集成后的软件功能与设计一致；

② 接口与设计一致；

③ 集成测试发现的问题得到修改并通过回归测试；

④ 满足集成测试终止条件；

⑤ 集成测试报告通过评审。

集成测试策划完成时，应编写软件集成测试计划。集成测试计划是否合理可行、是否满足充分性要求是十分值得关注的，因此应对软件集成测试计划进行评审。评审的主要内容包括：

(1) 测试人员组成合理、分工明确；

(2) 明确集成测试策略，集成测试策略恰当；

(3) 集成测试方法可行；

(4) 测试通过的准则满足软件质量、进度要求；

(5) 根据研制任务书、软件设计文档和其他软件质量要求明确需要进行测试的软件功能、性能和接口；

(6) 测试资源满足集成测试需求。

## 5.8.2　测试设计与实现

集成测试设计与实现阶段的主要工作是依据软件设计文档和集成测试计划，进行集成测试用例的设计，并完成必要的集成测试环境的建立和验证。集成测试设计与实现阶段完成时应编写集成测试说明，以及集成测试所需测试环境和测试环境验证报告。

集成测试与后续介绍的配置项测试和系统测试的区别在于软件测试是在集成的同时完成测试工作的。因此，需要对软件的结构、软件研制的进度等有充分的了解，在此基础上，根据质量和进度要求制定可行的集成顺序和测试执行顺序，这些都是影响软件集成测试用例设计的重要因素。

根据作者的经验，对于只有一个配置项的较小规模的软件，若开发人员较少，并基于一个开发平台进行开发时，可采用在充分的单元测试基础上，一次集成后直接进行配置项测试。

**1. 设计集成测试用例**

在进行集成测试用例设计时应遵循如下原则：

(1) 所有公共的接口都应有正常和异常的测试用例；

(2) 每一步集成后都应对集成后的模块进行功能、性能测试；

(3) 集成测试接口覆盖率应达到 100%；

(4) 接口测试用例应覆盖所有接口格式测试；

(5) 接口测试用例应覆盖对数据异常情况的测试。

**2. 建立集成测试环境**

集成测试时往往会遇到目标运行环境还不完备的情况，甚至存在部分待集成的模块未开发完成的情况，这些都将影响到后续软件的集成。因此，建立集成测试环境时，需要考虑这些风险，尽早制定缓解措施和应急计划，以保证顺利地开展集成和测试工作。

如果在条件允许的情况下，集成测试环境应尽可能使用目标运行环境以保证测试的有效性。若确实无法使用真实环境，建立的仿真环境需要经过认真地分析和确认，保证与真实环境的等效性。

**3. 测试就绪评审**

为了保证集成测试的充分性需要对集成测试计划、说明和测试环境就绪情况进行评审，评审的内容包括：

（1）集成测试策略和集成顺序是否恰当；

（2）每次集成后测试类型和测试项是否充分；

（3）测试项是否包括了测试终止要求；

（4）是否充分考虑了集成测试可能的风险，其缓解和应急计划是否恰当；

（5）测试说明是否完整、正确和规范；

（6）测试设计是否完整和合理；

（7）测试用例是否可行和充分；

（8）通过比较测试环境与软件真实运行的软件和硬件环境的差异，审查测试环境要求是否正确合理、满足测试要求；

（9）文档是否符合规范。

## 5.8.3 测试执行

集成测试执行活动是依据集成测试计划、测试说明按集成顺序进行集成和测试，记录实际的测试结果。当发现问题时应进行分析，如果是测试的问题应根据实际情况调整测试计划和说明，补充相应的测试用例；如果是软件的问题应如实、详细地记录测试结果，并提交问题报告。问题报告中应详细描述问题现象，分析问题类型和级别，给出改进意见建议，为后续的问题定位和解决提供支持。

如果软件进行了更改，不论是对集成测试中发现问题进行的更改，还是软件接口方面的更改都应进行相应的回归测试。回归测试的详细内容参见后续章节的介绍。

## 5.8.4 测试总结

集成测试完成后应对集成测试工作进行总结，以便评估集成测试中的问题是否得到有效解决，集成测试工作是否满足充分性要求。测试结果分析总结应写入软件测试报告，并应进行测试报告评审。

**1. 集成测试结果分析总结**

集成测试结果分析总结的内容包括以下 4 点。

（1）对集成测试过程进行总结。应对集成测试策划、集成测试设计与实现、集成测试执行过程进行总结，说明在各过程中开展的主要工作、参与人员和工作完成情况。

（2）对集成测试方法进行说明。应说明集成测试所采用的测试方法与策略，并说明采用这些方法的依据。

（3）对集成测试环境进行分析。应说明集成测试所使用的测试环境，包括测试工具、桩模块和驱动模块的使用情况，并对测试环境的差异性进行分析，说明测试环境是否满足集成测试的要求。

（4）对集成测试结果进行分析。集成测试结果的分析应包括对测试执行过程以及所有回归测试的情况的分析。主要内容包括：

① 测试时间；

② 测试人员；

③ 测试用例执行情况，内容包括测试用例数、通过的测试用例、未通过的测试用例、完全执行的测试用例、未完全执行的测试用例、未执行的测试用例；

④ 测试覆盖情况，包括对功能、性能、接口等覆盖情况，说明是否满足测试充分性要求；

⑤ 说明集成测试过程中发现的问题，并对问题解决情况进行说明；

⑥ 如果有遗留问题应说明可能的解决措施；

⑦ 对软件的整体情况进行评价，并提出改进的意见及建议。

**2．集成测试总结评审**

集成测试总结评审的内容包括：

（1）审查测试文档与记录内容的完整性、正确性和规范性；

（2）审查测试活动的有效性；

（3）审查测试环境是否符合测试要求；

（4）审查软件测试报告与软件测试原始记录和问题报告的一致性；

（5）审查实际测试过程与测试计划和测试说明的一致性；

（6）审查测试说明评审的有效性，例如是否评审了测试项选择的完整性和合理性、测试用例的可行性和充分性；

（7）审查测试结果的真实性和准确性；

（8）审查如果存在遗留问题，是否有处理措施。

# 第6章 CHAPTER 6

# 配置项测试

软件配置项测试是在软件配置项集成测试的基础上进行的，软件配置项集成测试完成了软件单元到配置项的集成测试。一般情况下，软件配置项测试可作为软件确认测试进行。软件配置项测试根据软件配置项的研制任务书、软件需求规格说明、用户手册等进行测试，确保软件配置项满足用户的要求，符合软件需求规格说明中定义的各项需求，同时，审查软件研制任务书、软件需求规格说明、软件实体、软件用户手册是否一致。

## 6.1 概述

### 6.1.1 配置项测试的定义

软件配置项测试的对象是软件配置项。根据 GJB 2786A—2009，软件配置项是满足最终使用要求并由需方指定进行单独配置管理的软件集合。计算机软件配置项的选择基于对下列因素的权衡：软件功能、规模、宿主机或目标计算机、开发方、保障方案、重用计划、关键性、接口考虑、需要单独编写文档和控制以及其他因素。

配置项测试一般情况下，应依据软件研制任务书、需求规格说明、接口需求规格说明和用户手册开展功能、性能、接口、安装、人机交互界面测试等。必要时，可包括文档审查、代码审查、静态分析、逻辑测试、强度测试、余量测试、安全性测试、恢复性测试、边界测试、数据处理测试和容量测试等。

### 6.1.2 配置项测试的目的

软件配置项测试的目的主要包括：

（1）确认该软件配置项是否达到了软件研制任务书、软件需求规格说明、软件接口需求规格说明等规定的各项要求；

（2）确定是否可以进行软件配置项验收交付和参加后续的系统集成测试。

软件研制方应根据软件研制任务书和软件需求规格说明中定义的全部需求及软件配置项测试计划，开展软件配置项测试工作。

软件配置项测试与软件集成测试的区别见 5.1.2 节中的说明。

软件配置项与单元测试存在较为明显的差别，主要表现在以下 5 个方面。

(1) 测试对象不同。配置项测试的对象是软件配置项，而单元测试的对象是软件单元。

(2) 测试的依据不同。配置项测试的测试依据是软件研制任务书和(或)需求规格说明、接口需求规格说明和用户手册，而软件单元测试的测试依据是软件设计文档。

(3) 测试环境不同。单元测试在开发环境下进行即可，同时需要建立桩模块和驱动模块；而配置项测试一般情况下，需要在目标环境下进行，并且需要开发相应的仿真程序和数据捕获程序。

(4) 测试策略不同。软件单元测试需要根据软件设计建立相应的测试策略，而软件配置项测试则一般采用黑盒测试方法。

(5) 测试内容存在差异。软件单元测试重点关注的是软件单元完成的功能，而配置项测试则需要从软件的方方面面考察软件研制任务书和(或)软件需求规格说明中描述的功能、性能、接口(配置项外部接口)、安全性、恢复性和安装性等是否符合要求。

### 6.1.3 配置项测试的重要性

在大型工程任务中，软件配置项作为研制活动管理的基本单元，从软件开发管理到技术活动的开展都以此为基础。因此，软件配置项测试在整个软件生命周期模型中发挥着非常重要的作用。突出表现在以下 5 个方面。

(1) 第一次完整地提交软件各项需求的实现。

(2) 通过配置项测试可以发现软件是否遗漏了软件研制任务书和(或)软件需求规格说明中的需求。

(3) 配置项测试期间软件用户会以不同的形式参与，例如测试就绪评审、测试执行等。可以通过配置项测试，使用户对软件完成的功能、性能等进行确认，保证软件交付的顺利进行。

(4) 配置项测试更多地关心软件实现各项需求的外在表现，即用户的期望。因此，配置项测试可以发现与其他配置项的接口、用户使用界面、性能、安全性、安装性等与用户直接相关的问题。避免在交付后，或参与系统集成时出现大量问题，造成进度、成本和质量方面的重大问题。

(5) 配置项测试环境较系统测试环境容易建立。由于配置项测试和系统测试都需要尽可能地在目标环境下进行，因此对单一的配置项测试环境而言比较容易实现。而整个系统涉及的各种软/硬件平台较为复杂，难以保证，往往到研制阶段后期才能到位，因此可通过配置项测试较早的在目标环境下确认部分系统的功能、性能等。

## 6.2 配置项测试原则

为保证软件配置项测试充分和有效，在进行软件配置项测试时，应遵循以下原则：

(1) 软件配置项测试的依据是软件需求规格说明和(或)软件研制任务书，需要时还应

包括接口需求规格说明和软件用户手册，后续描述中将配置项测试依据统称为软件需求；

（2）应逐项测试软件需求规定的软件配置项的功能、性能、接口等所有功能和非功能需求；

（3）测试软件需求覆盖率应达到100％，如果存在无法覆盖的情况，应说明原因，并制定后续处理的措施；

（4）必要时，在高层控制流图中作结构覆盖测试；

（5）软件配置项测试的环境应尽可能与软件目标环境一致，若不一致需要对差异性进行分析；

（6）测试用例的输入应至少包括有效等价类值、无效等价类值和边界值；

（7）测试人员应尽可能的与开发人员分离；

（8）针对软件配置项的具体需求应增加相应的专门测试。

## 6.3 配置项测试环境

配置项测试一般需要在目标环境下进行。若无法使用实际的运行环境时，可以使用有效的仿真环境进行配置项测试。但是，应对仿真环境与实际运行环境的差异进行分析，确保仿真环境与实际运行环境是等效的。另外，配置项测试需要使用一些专用工具，甚至需要开发一些数据仿真和数据获取工具才能完成相应的测试。

配置项测试环境的建立可以从以下4个方面考虑。

### 1. 硬件环境

在进行配置项测试时，应尽可能地使用软件实际的硬件环境。若确实无法保证环境的一致性，可搭建仿真环境来实现测试环境，但应对仿真环境与实际硬件环境的差异进行分析，说明对测试结果的影响。

### 2. 操作系统环境

在进行配置项测试中需要充分考虑操作系统的要求。

### 3. 数据库环境

目前以数据库为基础的信息系统非常普遍，因此在进行配置项测试时，需要根据软件研制任务书或软件需求规格说明中描述的数据库环境要求建立配置项测试环境。这是因为不同的数据库系统的性能、容量都存在不同，在配置项测试时要尤其关注这部分内容。

### 4. 网络环境

网络环境是影响软件性能等的关键因素，因此在进行配置项测试时，应按照研制要求采用实际的网络环境。某配置项测试的网络环境示意图如图6-1所示。

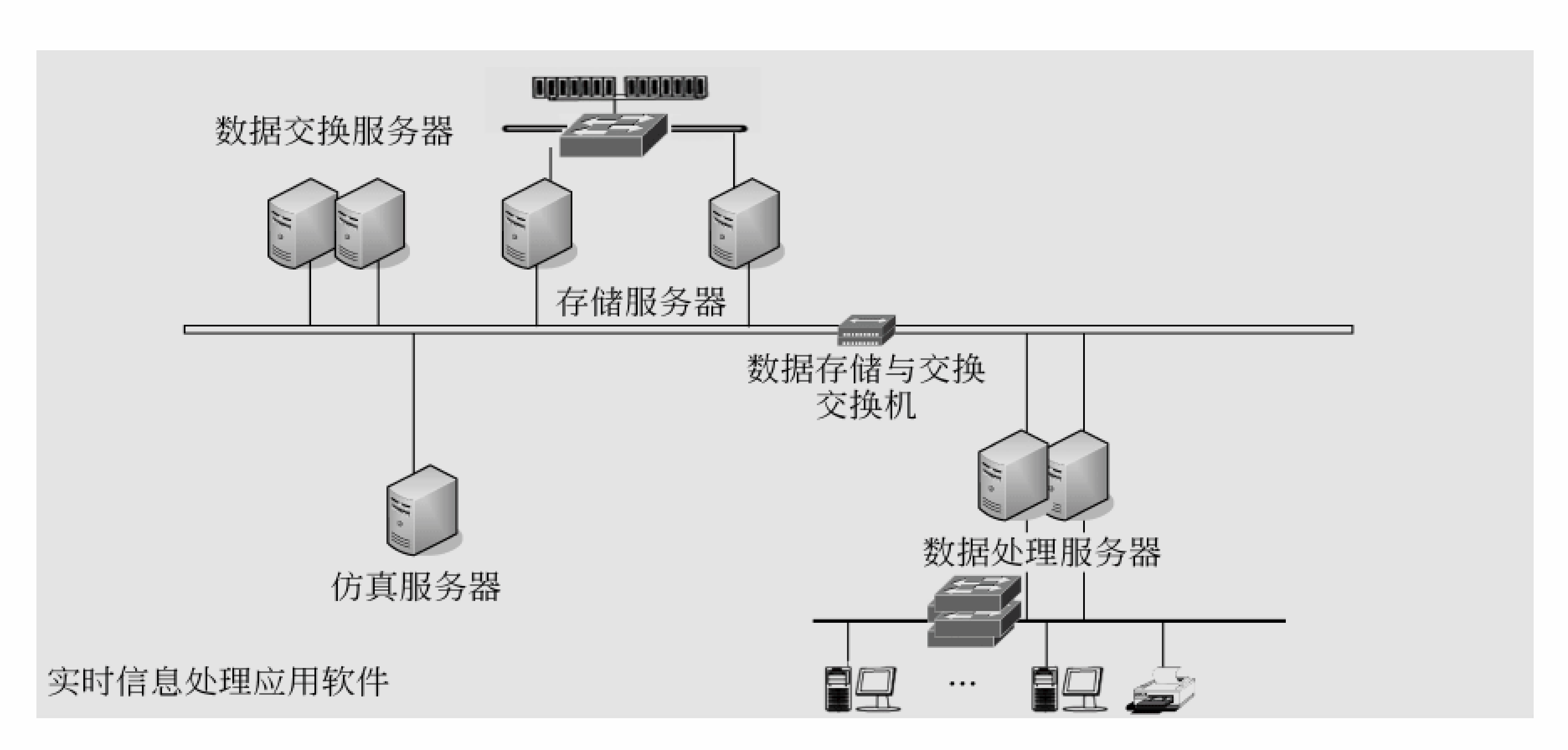



图 6-1　配置项测试网络环境示意图

## 6.4　配置项测试策略

配置项测试主要采用黑盒测试技术，对安全关键等级较高的软件，应适当地采用白盒测试技术保证测试的充分性。配置项测试策略需要根据被测软件的特点，确定测试仿真程序所需要提供的能力，并提出测试数据的要求。

进行软件配置项测试应具备以下基本条件：

(1) 具有软件研制任务书和(或)软件需求规格说明(含接口需求规格说明)、用户手册/操作手册；

(2) 已完成软件配置项的单元测试和集成测试；

(3) 软件配置项已按照软件配置管理要求进行受控管理；

(4) 软件配置项源代码通过编译或汇编；

(5) 软件配置项通常需要进行功能测试、性能测试、接口测试、人机交互界面测试、强度测试、余量测试、安全性测试、恢复性测试、边界测试、数据处理测试、安装性测试和容量测试，具体的测试类型需根据软件需求确定。

## 6.5　配置项测试内容

配置项测试的内容应根据软件研制任务书和(或)需求规格说明(含接口需求)中的要求确定，下面简称软件需求。一般情况下，应包括如下内容：

(1) 软件需求中定义的功能需求；

(2) 软件需求中定义的性能需求；

(3) 软件配置项的人机交互界面提供的操作和显示界面的正确性要求；

(4) 应测试运行在边界状态和异常状态下，或在人为设定的状态下，软件配置项的功能

和性能；

（5）应按软件需求的要求，测试配置项的安全性和数据的安全保密性；

（6）应测试配置项的所有外部输入、输出接口（包括和硬件之间的接口）；

（7）应测试配置项的全部存储量、输入/输出通道的吞吐能力和处理时间的余量；

（8）应按软件需求的要求，对配置项的功能、性能进行强度测试；

（9）应测试设计中用于提高配置项的安全性和可靠性的方案，如结构、算法、容错、冗余、中断处理等；

（10）对安全性关键的配置项，应对其进行安全性分析，明确每一个危险状态和导致危险的可能原因，并对此进行针对性地测试；

（11）对有恢复或重置功能需求的软件配置项，应测试其恢复或重置功能和平均恢复时间，并且对每一类导致恢复或重置的情况进行测试。

## 6.6 配置项测试方法

软件配置项测试主要采用黑盒测试技术，适当运用白盒测试技术。例如，对安全关键等级较高的嵌入式软件进行配置项级测试时，要辅助使用相应的白盒测试工具进行覆盖率统计和测试覆盖分析，帮助进行测试用例设计。

对安全关键等级较高的软件配置项测试，可能还包括文档审查、代码审查、静态分析、代码走查和逻辑测试。其中，文档审查、代码审查、静态分析、代码走查和逻辑测试方法的详细内容，见第 2 章和第 3 章相关描述。

软件配置项的测试内容主要是以软件研制任务书和（或）软件需求规格说明描述的软件需求（功能和非功能）为依据。另外，对一些软件配置项的隐含需求也需要进行测试。

软件测试的类型有很多种分类方法，较为常用的有传统的分类方法和以质量特性来划分的方法。这两种分类方法存在一定的对应关系，如表 6-1 所示。本书按照传统的分类方法进行介绍，本章主要介绍软件配置项测试常用的测试类型，包括：功能测试、性能测试、接口测试、人机交互界面测试、强度测试、余量测试、安全性测试、恢复性测试、边界测试、数据处理测试、安装性测试和容量测试。

一般情况下，可靠性测试和互操作性测试、兼容性测试等在系统测试中进行，相关内容将在后续章节进行介绍。进行哪些类型的测试，需要根据软件的具体特点进行具体分析，不能一概而论。

**表 6-1　质量特性与传统分类方法的对应关系**

| 传统分类 \ 质量特性 | 功能测试 | 性能测试 | 接口测试 | 人机交互界面测试 | 强度测试 | 余量测试 | 可靠性测试 | 安全性测试 | 恢复性测试 | 边界测试 | 数据处理测试 | 安装性测试 | 容量测试 | 互操作性测试 | 兼容性测试 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 适合性 | ✓ | | | | | | | | | ✓ | ✓ | | | | |
| 准确性 | | ✓ | | | | | | | | | ✓ | | | | |

续表

| 传统分类 / 质量特性 | 功能测试 | 性能测试 | 接口测试 | 人机交互界面测试 | 强度测试 | 余量测试 | 可靠性测试 | 安全性测试 | 恢复性测试 | 边界测试 | 数据处理测试 | 安装性测试 | 容量测试 | 互操作性测试 | 兼容性测试 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 互操作性 | | | ✓ | | | | | | | | | | | ✓ | |
| 安全保密性 | | | | | | | | ✓ | | | | | | | |
| 成熟性 | | | | | ✓ | | ✓ | | | | | | | | |
| 容错性 | | | ✓ | | ✓ | | ✓ | ✓ | | ✓ | | | | | |
| 易恢复性 | | | | | | | | | ✓ | | | | | | |
| 易理解性 | ✓ | | | ✓ | | | | | | | | | | | |
| 易学性 | ✓ | | | ✓ | | | | | | | | | | | |
| 易操作性 | ✓ | | | ✓ | | | | | | | | | | | |
| 吸引性 | | | | ✓ | | | | | | | | | | | |
| 时间特性 | | ✓ | | | | ✓ | | | | | | | ✓ | | |
| 资源利用性 | | ✓ | | | | ✓ | | | | | | | ✓ | | |
| 易分析性 | ✓ | | | ✓ | | | | | | | | | | | |
| 易改变性 | ✓ | | | ✓ | | | | | | | | | | | |
| 稳定性 | ✓ | | | | | | | | | | | | | | |
| 易测试性 | ✓ | | | | | | | | | | | | | | |
| 适应性 | ✓ | | | | | | | | | | | ✓ | | | |
| 易安装性 | | | | | | | | | | | | ✓ | | | |
| 共存性 | | | | | | | | | | | | | | | ✓ |
| 易替换性 | | | | | | | | | | | | | | | ✓ |

## 6.6.1　功能测试

功能测试是软件配置项测试中最基本的测试，主要是依据软件需求规格说明中的功能需求进行的测试，以确认其功能是否满足要求。

**1. 测试技术要求**

功能测试一般需进行下列各项的测试，其中(1)、(2)、(3)项为必做项：

(1) 使用正常值等价类进行测试；

(2) 使用异常值等价类进行测试；

(3) 对每个功能使用合法边界值和非法边界值进行测试；

(4) 用一系列真实的数据进行超负荷、饱和及其他“最坏情况”的测试；

(5) 在配置项测试时，应对配置项控制流程的正确性、合理性等进行测试。

**2. 实施要点**

配置项功能测试一般是基于软件需求规格说明的测试，应对软件需求规格说明进行分析，梳理出测试需求项，确定测试输入数据和输出数据，提出可能需要的数据准备要求、测试工具和需开发的测试程序等要求，具体的步骤如下。

（1）根据软件需求规格说明标识需测试的每一项功能需求，包括隐含的功能需求。在功能测试中，需要特别关注隐含需求的梳理，根据作者们从事软件测试多年的经验，研制人员在需求规格说明中明确描述的软件功能需求一般最多达到80%。而出现问题最多的地方往往涉及隐含需求。

（2）确定每一项功能应满足的要求。

（3）分析每一项功能可能出现的异常情况。

（4）对功能测试需求划分优先级。如果软件需求规格说明中已明确各项功能的优先级，测试需求的优先级应与需求规格说明中的相应优先级保持一致；否则，可以根据该功能失效后对软件造成的影响确定测试需求的优先级。

（5）对每项功能测试需求应确定测试数据输入和测试结果捕获方法。

（6）功能测试中最常用的测试设计方法是等价类划分方法，包括有效等价类和无效等价类。有效等价类用于正常工作流程、正常输入值测试；无效等价类用于非正常工作流程、非正常值输入测试。

（7）边界值分析方法是功能测试中对等价类划分方法的重要补充。很多情况下，软件在处理边界值时经常会发生错误，因此针对边界进行分析、测试十分必要。

（8）因果图、决策表、基于场景的测试、组合测试和猜错法等第3章中介绍的动态测试用例设计方法，可以根据软件配置项实现的具体功能适当地加以应用。

## 6.6.2 性能测试

性能测试是对软件需求规格说明中规定的性能需求逐项进行的测试，以验证其性能是否满足要求。

**1. 测试技术要求**

性能测试一般需进行下列各项的测试，其中(1)、(2)、(3)项为必做项：

（1）软件在定量结果计算时的处理精度测试；

（2）软件时间特性和实际完成功能的时间（响应时间）测试；

（3）软件完成功能所处理的数据量测试；

（4）软件运行所占用空间的测试；

（5）软件负荷潜力测试。

**2. 实施要点**

（1）按照软件需求规格说明标识需要测试的性能需求，额外增加的性能测试需求应得到用户的确认，并且性能指标要适当考虑软件的应用环境和任务要求。

（2）测试处理精度时，可通过捕获输出数据确认软件的处理精度是否满足要求。

（3）测试软件响应时间时，可通过记录处理前时间T1和处理后时间T2，计算处理后与

处理前时间之差获得软件响应时间。

（4）测试软件数据处理周期和数据量时，可按照软件需求规格说明中要求的速度或数据量发送数据，捕获处理后输出的数据正确且无丢失即可认为满足要求。

（5）对于时间指标的测试，需要使用相匹配的测量设备，根据需要可在时统时间、计算机时间、手持秒表等设备中选取。

（6）当时间指标要求高于 1s 时，应编写测试程序获得时统时间或者计算机时间作为计算时间。

（7）由于测试的不确定性，性能测试用例应执行多次，应准确、详细地记录实际的执行结果，并进行最大值、最小值、平均值等分析。

（8）性能测试时，应考虑在正常、最好、最坏情况下的性能差异。

（9）与硬件环境相关的性能测试应在目标环境下实施。

## 6.6.3　接口测试

接口测试是对软件需求规格说明或设计文档的接口需求逐项进行的测试。

**1. 测试技术要求**

接口测试一般需进行下列各项的测试，其中（1）、（2）项为必做项：

（1）测试所有软件配置项的外部接口，包括与其他软件配置项和硬件配置项之间的接口，检查接口信息的格式及内容是否满足要求；

（2）对每一个外部输入/输出接口必须进行正常和异常情况的测试。

**2. 实施要点**

（1）配置项接口测试的依据是软件需求规格说明中的外部接口定义或软件接口需求规格说明中定义的接口。

（2）对输入接口进行测试时，应按照接口信息的格式和内容，使用测试程序输入格式正确、内容正确的测试数据，以及格式错误、内容错误的测试数据。

（3）对输出接口进行测试时，应使用测试程序捕获被测软件的输出数据，检查是否满足接口信息的格式要求，内容是否正确。

（4）对 API 接口进行测试时，需要关注是否支持多个调用的情况。

（5）对 TCP、串口类接口进行测试时，应模拟几帧粘连在一起的情况，测试应用软件是否能从粘连的数据中提取有效数据。

（6）对网络接口进行测试时，应关注如下错误类型的测试：

① 任务标志错误，包括：不存在的任务标志或非本次任务标志；

② 信源信宿错误，包括：不存在的信源信宿或非规定的信源信宿；

③ 数据标志错误，不存在的数据标志或非规定的数据标志；

④ 包序号错误，包括：包序号不连续、包序号倒序、包序号重复；

⑤ 数据域错误，包括：数据域长度字段值小于实际数据域长度、数据域长度字段值大于实际数据域长度等。

（7）对以文件方式定义的接口进行测试时，错误一般应包括文件不存在、文件打开失

败、文件保存失败、文件中数据不符合要求、文件中数据字段不完整等。

（8）对数据库接口进行测试时，应对下列内容进行测试：

① 数据库连接异常及恢复；

② 大规模并发访问控制；

③ 数据表增、删、改权限控制；

④ 数据库同步操作；

⑤ 数据库备份及还原；

⑥ 数据标识唯一性判别；

⑦ 数据表元素修改和插入的不完整提交；

⑧ 数据元素完整性判别；

⑨ 异常数据元素字段写入和修改控制；

⑩ 数据表间一致性检查；

⑪ 数据元素修改和删除的依赖控制；

⑫ 数据表键值设计合理性检查等。

## 6.6.4 人机交互界面测试

人机交互界面测试是对所有人机交互界面提供的操作和显示界面进行的测试，以检验是否满足用户的要求。

**1. 测试技术要求**

人机交互界面测试一般需进行下列各项的测试，其中（2）、（3）两项中必做一项：

（1）测试操作和显示界面及界面风格与软件需求规格说明、用户手册或操作手册中要求的一致性和符合性；

（2）以非常规操作、误操作、快速操作来检验人机界面的健壮性；

（3）测试对错误命令或非法数据输入的检测能力与提示情况；

（4）测试对错误操作流程的检测与提示；

（5）对照用户手册或操作手册逐条进行操作和观察。

**2. 实施要点**

（1）人机交互界面测试的依据是软件用户手册或操作手册。

（2）对照用户手册或操作手册逐条进行操作和检查。

（3）通过界面输入错误的和无效的参数，例如对需要输入整数的界面操作，可输入字符、浮点数等非法值，测试软件是否有相应的验证和防范措施。

（4）在进行是否设置默认值的测试时，可在界面中不输入任何数值或字符，测试软件是否采用默认值进行后续工作。

（5）在进行违反流程的测试中，应采用用户使用过程中可能出现的违反操作流程的情况进行测试，测试软件是否具有防止误操作的能力。

（6）使用界面测试工具，编写界面快速操作和重复操作的脚本，测试软件是否具有快速响应能力或具有防止重复操作的机制。

(7) 测试对重要信息进行不可逆操作(例如删除等操作)时,是否有防范误操作手段或提示确认操作的信息。

### 6.6.5 强度测试

强度测试是强制软件运行在不正常到发生故障的情况下(设计的极限状态到超出极限),检验软件可以运行到何种程度的测试。

**1. 测试技术要求**

一般根据软件配置项的具体需求选择进行如下强度测试:

(1) 提供最大处理的信息量;

(2) 提供数据能力的饱和实验指标;

(3) 提供最大存储范围(如常驻内存、缓冲、表格区、临时信息区);

(4) 在能力降级时进行测试;

(5) 在人为错误(如寄存器数据跳变、错误的接口)状态下进行软件反应的测试;

(6) 通过启动软件过载安全装置(如临界点警报、过载溢出功能、停止输入、取消低速设备等)生成必要条件,进行计算过载的饱和测试;

(7) 需进行持续一段规定的时间,而且连续不能中断的测试。

**2. 实施要点**

(1) 一般情况下,强度测试与软件配置项的性能要求有较为紧密的关系,因此,可以针对性能要求考虑对软件配置项的强度进行相应的测试。

(2) 强度测试重点考察软件在运行环境最为复杂的情况下,完成相应功能的能力,因此需要设计软件在复杂情况下所需的环境。

(3) 对与数据流量相关的强度测试时,首先输入正常数据量,然后逐步提高数据量使其达到性能指标所要求的数据量,观察被测软件输出是否正常,在该数据量下运行所要求的时间后,继续提高数据量以达到性能下降的临界状态,记录临界状态下的数据量。继续提高数据量,使得被测软件处于降级处理状态,随后降低数据量,使被测软件恢复正常。在测试过程中,应关注被测软件的 CPU、内存占用及其他相关的性能指标情况。

(4) 在进行软件长时间连续运行的测试时,时间应以软件需求规格说明或其他相关文档中要求的为准。没有明确要求的,默认为一个业务周期。在连续运行过程中,应达到最大处理能力并略超出一点,再恢复到正常处理水平,重复操作多次。

### 6.6.6 余量测试

余量测试是对软件是否达到需求规格说明中要求的余量的测试。如果没有明确要求时,一般至少保留 20%的余量。

**1. 测试技术要求**

一般根据软件配置项的具体需求选择进行如下余量测试:

(1) 全部存储量的余量;

（2）输入/输出及通道的吞吐能力余量；

（3）功能处理时间的余量。

**2. 实施要点**

（1）余量测试一般与功能、性能、强度等测试一起进行。

（2）应注意观察软件在功能、性能、强度等测试中，处于空闲、正常、满负荷、临界状态下的 CPU 和内存的使用情况，并计算出余量。

（3）在进行处理时间的余量测试时，使得软件保持正常运行状态，观察功能处理时间，与软件需求规格说明中要求的功能处理时间进行比较，计算出余量。

（4）在进行输入/输出及通道的吞吐能力余量测试时，可通过输入最大数据量，观察被测软件的输出，与软件需求规格说明中要求的输入/输出及通道吞吐能力进行比较，计算出相应的余量。

（5）软件配置项余量测试一般应在软件的目标运行环境下进行，若采用仿真环境应充分分析其差异性，以便保证测试结果的有效性。

（6）同性能测试相同，余量测试也需要进行多次测试。

### 6.6.7 安全性测试

安全性测试是检验软件中已存在的安全性、安全保密性措施是否有效的测试。测试应尽可能在符合实际使用的条件下进行。

**1. 测试技术要求**

一般根据软件配置项的具体需求选择进行如下安全性测试：

（1）对安全关键等级较高的软件配置项，必须单独测试安全性需求；

（2）在测试中全面检验防止危险状态措施的有效性和每个危险状态下的反应；

（3）对设计中用于提高安全性的结构、算法、容错、冗余及中断处理等方案，必须进行针对性测试；

（4）对软件处于标准配置下其处理和保护能力进行测试；

（5）应进行对异常条件下系统/软件的处理和保护能力的测试（以表明不会因为可能的单个或多个输入错误而导致不安全状态）；

（6）对输入故障模式的测试；

（7）必须包含边界、界外及边界结合部的测试；

（8）对“0”、穿越“0”以及从两个方向趋近于“0”的输入值的测试；

（9）必须包括在最坏情况配置下的最小输入和最大输入数据率的测试；

（10）对安全性关键的操作错误的测试；

（11）对具有防止非法进入软件并保护软件的数据完整性能力的测试；

（12）对双工切换、多机替换的正确性和连续性的测试；

（13）对重要数据的抗非法访问能力的测试。

**2. 实施要点**

（1）对关键等级较高的软件配置项进行软件安全性测试时，应基于软件安全性分析的

基础开展。

（2）安全性测试内容应覆盖所有安全性需求。

（3）在进行软件安全性测试时，应考虑各种异常输入和异常操作。

（4）对安全关键的操作进行测试时，需要测试其是否提供再次确认操作。

（5）在进行双工切换操作时，需要考虑以下测试内容：

① 重复多次双工切换、多机替换；

② 模拟无主机的状况，测试软件是否能报警或自动选出一个主机；

③ 模拟多主机的状况，测试软件是否能报警或自动选出一个主机，并且在多主机期间没有产生违反安全性的后果。

（6）测试用错误用户名、错误密码、超出权限等非法身份访问软件。

（7）应检查软件在进行权限判断时，是否无信息泄露。

（8）对于可远程提供 SQL 查询语句的软件，应测试其防止“SQL 注入”攻击的能力。对于可远程提供命令行执行语句的软件，应测试其防止“Shell 命令注入”攻击的能力。

（9）对有用户权限管理的软件，除了应进行各类用户权限管理测试外，还应检查保存用户密码的数据库或文件是否进行了加密保存，对安全关键数据是否进行了加密处理。

（10）测试各种资源不满足的情况，软件是否能够避免崩溃或异常退出。例如文件访问操作中路径不存在、文件不存在、网络应用中网卡禁用/不存在、串口通信软件找不到串口设备等。

### 6.6.8　恢复性测试

恢复性测试是对有恢复或重置功能的软件的每一类导致恢复或重置的情况，逐一进行的测试，以验证其恢复或重置的能力。恢复性测试是要证实在克服硬件故障后，系统能否正常地继续进行工作，且不对系统造成任何损害。

**1. 测试技术要求**

一般根据软件配置项的具体需求选择进行如下恢复性测试：

（1）探测错误功能的测试；

（2）能否切换或自动启动备用硬件的测试；

（3）在故障发生时能否保护正在运行的作业和系统状态的测试；

（4）在系统恢复后，能否从最后记录下来的无错误状态开始继续执行作业的测试。

**2. 实施要点**

（1）软件恢复性测试应按照软件需求规格说明中定义的软件恢复性要求进行。

（2）嵌入式软件“看门狗”测试被认为是恢复性测试中的典型类型。在进行“看门狗”测试时，可修改被测软件代码，加入死循环代码。通过引发死循环，测试软件在这种情况下，是否能够通过看门狗复位使程序重新启动。

（3）软件断点续传功能的测试被认为是较为典型的恢复性测试。

（4）对具有数据恢复能力的软件，测试在断电等异常情况发生时，软件重新运行后恢复运行的能力。

### 6.6.9 边界测试

边界测试是对软件处在边界或端点情况下运行状态的测试。

**1. 测试技术要求**

一般根据软件配置项的具体需求选择进行如下边界测试：

(1) 软件的输入域和输出域的边界或端点的测试；

(2) 状态转换的边界或端点的测试；

(3) 功能界限的边界或端点的测试；

(4) 性能界限的边界或端点的测试；

(5) 容量界限的边界或端点的测试。

**2. 实施要点**

(1) 边界测试不仅要考虑输入域的测试，还需要进行输出域的测试。

(2) 边界测试一般需考虑小于下边界、等于下边界、大于下边界、小于上边界、等于上边界、大于上边界6种情况的测试。

(3) 对输出域的边界测试，应通过控制输入数据实现输出边界的测试。

(4) 性能界限的边界测试往往是强度测试的考虑内容，可一并考虑。

(5) 容量界限的边界测试与容量测试是一致的。

(6) 对于功能边界的测试，应建立达到功能边界的条件。例如超过某个边界时使用不同的测量设备，测试时就需要建立模拟输入，使系统刚好处于这种状态下，检查是否按照规定选择了正确的测量设备。

### 6.6.10 数据处理测试

数据处理测试属于功能测试的具体化，是对完成专门数据处理功能所进行的测试。

**1. 测试技术要求**

数据处理测试一般需进行以下(2)、(3)、(4)、(5)测试项中至少一项测试：

(1) 数据采集功能的测试；

(2) 数据融合功能的测试；

(3) 数据转换功能的测试；

(4) 剔除坏数据功能的测试；

(5) 数据解释功能的测试，例如：拆包、解包。

**2. 实施要点**

(1) 通常情况下，在软件需求规格说明中有专门的数据处理功能时，需进行数据处理测试。

(2) 在进行采集功能测试时，应进行如下测试：

① 数据采集周期是否与需求一致，特别是需对多种数据进行采集时各自不同的采集周期是否符合要求，另外还应对采样时间有中断、误差是否有积累等情况进行测试；

② 有多路数据时，数据的选择策略是否正确。

(3) 在进行数据融合功能的测试时，应进行如下测试：

① 应进行单个数据跳点处理方法的正确性测试；

② 应使用等价类方法，采用有效等价类、无效等价类和边界值进行测试；

③ 应对有误差修正要求的数据进行误差修正测试；

④ 应考虑过零时数据融合处理是否正确；

⑤ 应考虑部分数据异常时，数据融合处理是否正确；

⑥ 应考虑时间信号异常时，数据融合处理是否正确。

(4) 在进行数据转换功能的测试时，应进行如下测试：

① 量纲转换正确性测试；

② 时间制使用正确性测试，例如北京时或世界时、12 小时制或 24 小时制，特别关注过零点的处理；

③ 关键时间正确性判断方法测试，例如“三取二”或“七取四”判断方法；

④ 当软件使用多个坐标系进行数据处理时，应对不同坐标系下，数据转换正确性进行测试。

(5) 在进行数据解包功能的测试时，应进行如下测试：

① 数据帧格式的正确性测试，例如一帧只含一包完整数据、一帧含多包完整数据、多帧含一包完整数据等情况；

② 应考虑数据帧异常情况的测试，例如一帧中数据包数标识与实际数据包数不一致、数据包识别码与实际数据内容不一致等。

### 6.6.11 安装性测试

安装性测试是对安装过程是否符合安装规程的测试，以发现安装过程中的错误。

**1. 测试技术要求**

一般根据软件配置项的具体需求选择进行如下安装性测试：

(1) 不同配置下的安装和卸载测试；

(2) 安装规程的正确性测试。

**2. 实施要点**

(1) 安装性测试应按照软件用户手册/操作手册中规定的安装规程进行，在被测软件要求的软/硬件配置下安装被测软件，并运行被测软件，验证安装后被测软件的各项功能运行正常之后才能确认安装功能正常。

(2) 卸载被测软件，验证卸载被测软件后是否影响其他软件的运行。完成卸载后，应重新进行安装，验证软件是否能够正确地被重新安装，并检查安装后各项功能是否能够正常运行。

(3) 应在安装前获得计算机当前应用程序清单，安装和卸载后检查应用程序清单，比对两个清单检查软件安装和卸载功能是否正确。

(4) 对于专用软件，应在目标计算机环境下执行安装和卸载测试。对于通用软件，需在

其支持的各种软件平台环境下进行测试，例如 Windows 系统，在 Windows XP、Windows 7、Windows Server 2008 以及 Windows 8 等环境下测试，必要时，还需在已安装有杀毒软件的环境下进行测试。

### 6.6.12 容量测试

容量测试是检验软件的能力最高能达到什么程度的测试。

**1. 测试技术要求**

容量测试一般应测试在正常情况下，软件所具备的最高能力，例如响应时间或并发处理个数等能力。

**2. 实施要点**

（1）一般情况下，容量是指最快的响应时间、最大的并发数量或最大的吞吐量等，在强度测试过程中获得的临界值即为软件的最高处理能力。

（2）容量测试一般结合性能测试、强度测试、余量测试等其他测试进行。

（3）容量测试的实施要点见强度测试的实施要点。

## 6.7 配置项测试用例设计

配置项测试一般采用黑盒测试方法，用例设计通常采用等价类划分、边界值分析、因果图、决策表、组合测试和错误推测等方法，有关详细内容参见第 3 章。

本节重点介绍对关键等级较高的软件进行测试时，采用基于安全性分析的测试方法。

### 6.7.1 概述

为了解决高可靠性安全性软件测试时，测试用例设计的充分性和有效性不足的问题，可采用基于安全性分析的测试用例设计及用例选取的方法。首先，利用软件失效模式和影响分析、故障树分析相结合的方法进行软件安全性分析。按照失效影响分析、失效模式危害分析的结果，选择影响大、危害程度高的失效影响和失效模式作为顶事件和中间事件建立软件故障树。然后，将故障树的最小割集作为测试用例设计模式全集，在测试资源有限的条件下，根据最小割集的重要程度建立测试用例设计模式的选择准则。最后，依据选定的测试用例设计模式中每个事件的特点完成测试用例的详细设计。

大型工程中对软件等级的划分都是以安全等级进行划分的，因此对该类软件的测试也需要在安全性分析的基础上开展才能保证其充分性、有效性。

配置项测试时采用的软件安全性分析方法一般选用软件失效模式和影响分析法(software failure mode and effect analysis，SFMEA)与软件故障树法(software fault tree analysis，SFTA)，使用这两种方法进行软件安全性分析时各有优缺点。SFMEA 是自底向上的分析方法，无法表达失效原因之间的逻辑关系。SFTA 是自顶向下树状结构的分析方

法，这种方法在选取顶事件时，可能会遗漏潜在的顶层失效事件。另外，SFTA 在分析故障原因时，也可能会有遗漏，这样将影响底事件的重要度排序。

因此，本节利用软件失效模式和影响分析、故障树分析相结合的方法，进行软件安全性分析。这两种方法的结合，弥补了软件失效模式和影响分析对失效原因分析不彻底的问题，也避免了软件故障树分析方法可能忽略关键顶事件的问题。在安全性分析的基础上，通过对获得的软件故障树进行定性和定量的分析，建立了测试用例设计模式的选取策略，在最小割集的基础上，采用该策略完成测试用例设计模式的生成。

采用 SFMEA 与 SFTA 相结合的方法——SFME&FTA 进行安全性分析，做到这两种方法的优势互补。在安全性分析的基础上，进行软件测试用例的设计，解决了高安全性要求软件测试的充分性和有效性问题。

## 6.7.2　SFME&FTA 综合分析

SFME&FTA 方法首先按照 SFMEA 分析软件的失效模式、失效原因、失效模式的影响以及失效影响程度，根据失效影响危害分析结果选择顶事件，将失效模式作为中间事件，进行 SFTA 分析。SFTA 分析结果作为测试用例设计模式。SFME&FTA 方法如图 6-2 所示。

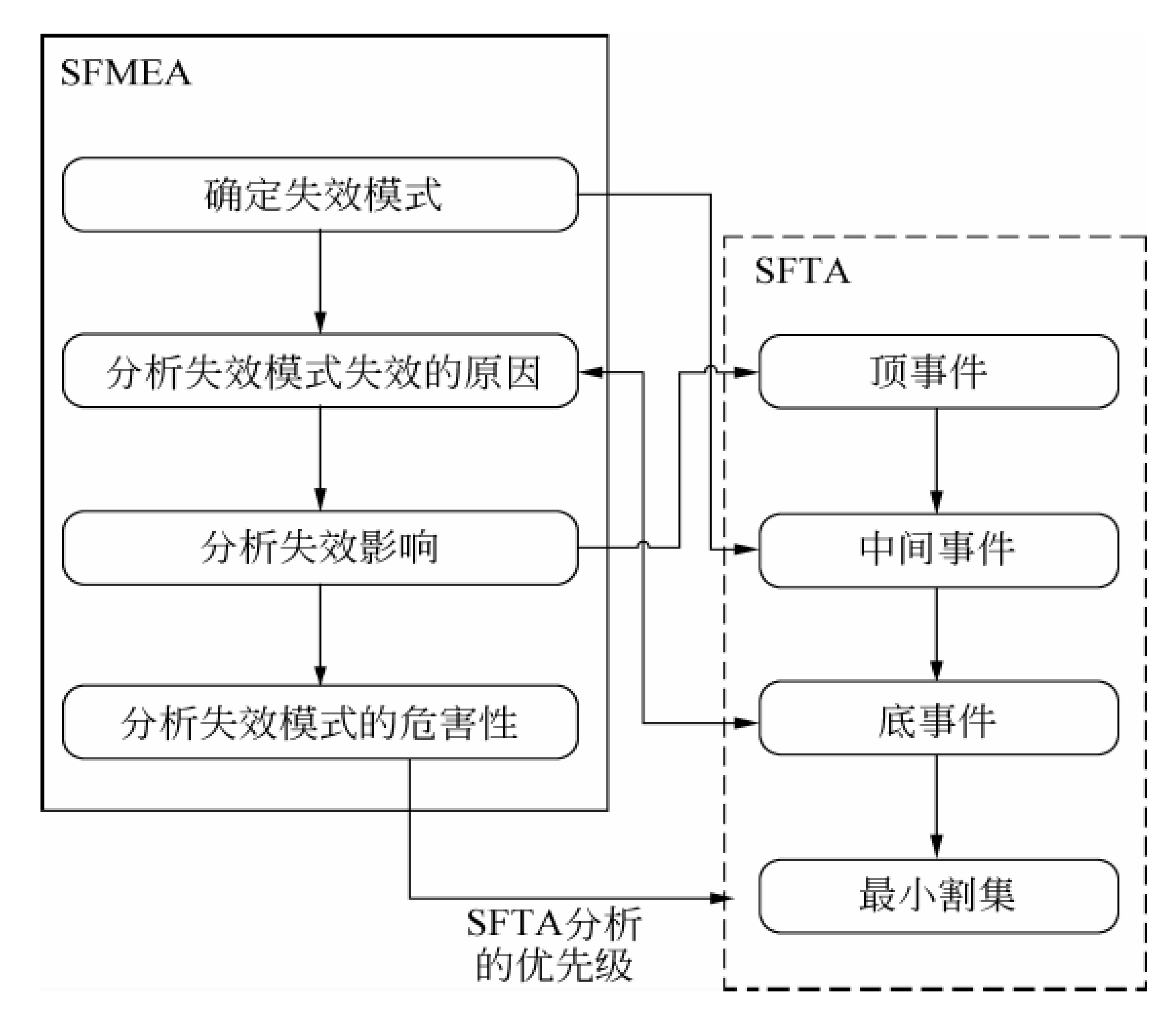



**图 6-2　SFME&FTA 方法示意图**

**1. 软件失效模式和影响分析**

SFME&FTA 方法首先对软件进行失效模式与影响分析。SFMEA 是分析软件系统中所有可能的软件失效模式及其对系统造成的影响，同时对每一种失效模式按照严重程度、概率以及测试难度对失效模式进行分类。

1）失效模式分析

在进行软件失效模式分析时，需要依据软件需求规格说明分析软件可能发生的失效。

一般情况下，软件都具有多个功能，在进行失效模式分析时，应考虑每个功能可能的失效模式。软件的失效模式一般包括运行时失效、输入失效、输出失效、性能失效等。详细的失效模式的示例，如表 6-2 所示。

**表 6-2　失效模式示例**

| 失效模式类型 | 失 效 模 式 |
|---|---|
| 输入失效 | (1) 未收到输入 |
| | (2) 收到的数据不正确 |
| | (3) 收到的数据不完整 |
| | (4) 收到的数据精度不满足要求 |
| 运行失效 | (1) 程序无法启动 |
| | (2) 程序运行中非正常中断 |
| | (3) 程序运行不能终止 |
| | (4) 程序不能退出 |
| | (5) 程序运行陷入死循环 |
| | (6) 程序运行对其他软件或环境产生有害影响 |
| 输出失效 | (1) 输出结果错误(例如输出项缺损或多余等) |
| | (2) 输出数据精度不满足要求 |
| | (3) 输出参数不完整 |
| | (4) 输出格式错误 |
| | (5) 输出提示信息不正确 |
| 未满足功能及性能的失效 | (1) 未达到功能/性能的要求 |
| | (2) 不能满足用户对运行时间的要求 |
| | (3) 不能满足用户对数据处理量的要求 |
| | (4) 多用户系统不能满足用户数的要求 |
| 其他失效 | (1) 程序运行改变了系统配置要求 |
| | (2) 程序运行改变了其他程序的数据 |
| | (3) 操作系统错误 |
| | (4) 硬件错误 |
| | (5) 整个系统错误 |
| | (6) 人为操作错误 |
| | (7) 接口故障 |
| | (8) I/O 定时不准确导致数据丢失 |
| | (9) 维护不合理/错误 |

2）失效原因分析

软件失效原因是软件中潜在的缺陷，软件失效可能由一个或多个软件缺陷引起的，在进行失效原因分析时，应尽可能全面地分析所有可能的软件失效原因，为软件测试用例设计提供尽可能准确的设计模式。

3）失效影响分析

失效影响分析是指每个失效模式对软件本身或其他软件/硬件系统的影响。软件失效影响的严重程度等级使用严酷度（effect severity ranking，ESR）来定义。严酷度等级的评分准则如表 6-3 所示。

**表 6-3　严酷度等级的评分准则**

| 软件失效模式影响的程度 | 评 分 等 级 |
|---|---|
| 影响系统运行的安全性，或不符合安全规定 | 5 |
| 影响系统丧失主要功能而不能运行 | 4 |
| 系统仍能运行，但运行水平降级，用户不满意 | 3 |
| 影响轻度 | 2 |
| 无影响 | 1 |

4）失效模式危害性分析

危害分析可以用失效模式的风险优先数 RPN 来表示，即失效影响严酷度等级与失效模式发生概率等级（occurrence probability ranking，OPR）的乘积。

（1）失效模式发生的概率等级的定义如表 6-4 所示。

**表 6-4　失效模式发生概率等级评分**

| 软件失效模式发生的可能性 | 评 分 等 级 |
|---|---|
| 故障模式发生的可能性非常高 | 5 |
| 故障模式发生的可能性高 | 4 |
| 故障模式发生的可能性中等 | 3 |
| 失效模式发生的可能性相对较低 | 2 |
| 失效模式发生的可能性极低 | 1 |

（2）测试难度等级的评分准则。由于测试资源（人员、设备和时间）是有限的，测试时需要选择关键的、重要的以及能够被测试的软件失效模式进行测试。在进行失效模式选择时，还需要对测试难度等级进行定义。测试难度等级（testing difficulty ranking，TDR）评定的准则如表 6-5 所示。

**表 6-5　测试难度等级评定准则**

| 测 试 难 度 | 评 分 等 级 |
|---|---|
| 被测试出的难度非常大 | 1 |
| 被测试出的难度大 | 2 |

续表

| 测 试 难 度 | 评 分 等 级 |
|---|---|
| 被测试出的难度中等 | 3 |
| 被测试出的难度小 | 3 |
| 被测试出的非常容易 | 5 |

**2. 软件故障树分析**

1）软件故障树顶事件和中间事件选择

（1）软件故障树顶事件的选择。在测试资源有限的情况下，测试人员不得不选择软件故障后造成影响较大的问题进行测试。这样，软件故障树顶事件可以根据影响程度分析结果进行选择。一般情况下，影响程度等级在3级以上（含3级）的失效模式影响应作为故障树的顶事件。

（2）软件故障树中间事件的选择。故障树的中间事件通过定义RPN与测试难度等级乘积的值来进行选择，也可以通过指定严酷度、发生概率和测试难度中的任意一个或任意两个的组合值作为失效模式选择的策略。

假设失效模式的全集为$\Omega$，对每一个失效模式$\mathrm{FM}_i \in \Omega$，其严酷度、发生概率和测试难度分别为$\mathrm{ESR}(\mathrm{FM}_i) \triangleq \mathrm{ESR}_i$、$\mathrm{OPR}(\mathrm{FM}_i) \triangleq \mathrm{OPR}_i$、$\mathrm{TDR}(\mathrm{FM}_i) \triangleq \mathrm{TDR}_i$，并令$R$表示它们的集合，即$R \equiv \{\mathrm{ESR}, \mathrm{OPR}, \mathrm{TDR}\}$。假设中间事件的选取标准函数为Rule(FM)，则中间事件的集合$I \subset \Omega$可表示为$I = \{\mathrm{FM} \in \Omega \mid \mathrm{Rule}(\mathrm{FM}) \geqslant c\}$，其中Rule(FM)可根据需要进行设定，一般地可定义为严酷度、发生概率和测试难度中的任意一个或多个的乘积，即$\mathrm{Rule}(\mathrm{FM}_i) \in \Theta \equiv \{r_p \mid r_p \in R\} \cup \left\{\prod_{p=1}^{3} r_p \mid r_p \in R\right\} \cup \{r_p \times r_q \mid r_p \in R, r_q \in R, q > p\}$，而$c$为根据Rule(FM)及实际需要设定的一个常数。

2）软件故障树的定性和定量分析

在通过上述(1)中失效模式影响选择策略获得影响大的软件失效影响作为故障树的顶事件，按照(2)中描述的方法将选择的失效模式作为故障树的中间事件，将失效原因作为底事件，形成软件故障树，利用Fuseell-Vesely算法，通过对软件故障树进行定性分析得到最小割集。

最小割集对顶事件发生的影响程度是不同的，在评估最小割集的重要度时，可用概率重要度和关键重要度来衡量。将软件故障树分析后的结果作为测试用例设计模式。

## 6.7.3 建立软件测试用例设计模式

将软件故障树定性分析结果的最小割集作为测试用例设计模式。

根据测试用例设计模式生成的测试用例的数量一般都是比较大的，为了实现测试资源有效的条件下选择关键的测试用例设计模式，需要确定测试用例设计模式的选择策略。在制定测试用例选择策略时，采用定性与定量相结合的方法。

（1）测试用例设计模式定性分析。在确定选择策略时，首先进行定性分析获得最小割集，在定性分析的基础上，可根据下列原则选择测试用例设计模式：

① 阶数越小的最小割集越重要，应将阶数最小的最小割集纳入测试用例设计模式集；

② 在不同最小割集中，重复出现次数越多的底事件越重要，应将这些底事件涉及的最小割集纳入测试用例设计模式集；

③ 在低阶最小割集中出现的底事件比高阶最小割集中的底事件重要，应将最小阶数最小割集中的底事件涉及的最小割集纳入测试用例设计模式集。

(2) 测试用例设计模式定量分析。测试用例设计模式选择策略，可以对最小割集的重要程度进行定量分析，通过分析故障树中每个底事件在故障树中的结构重要度，进行测试用例设计模式的选择。每个底事件在故障树中的结构重要度计算方法如下：

$$I_i^\phi = \sum_{2^{n-1}} [\Phi(1_i, X) - \Phi(0_i, X)] / 2^{n-1}$$

其中，$I_i^\phi$ 为第 $i$ 个根节点的重要度，$n$ 为所含节点的数量，$\Phi(1_i, X)$ 为第 $i$ 个节点为“1”时系统的状态，$\Phi(0_i, X)$ 为第 $i$ 个节点为“0”时系统的状态。

## 6.7.4　应用实例

(1) 获取软件故障树。在某软件测试中，获得了如图 6-3 所示的软件故障树，其中中间事件“未关机”的发生是由 Ggjx1，Ggjx2，Ggjx3 中的任意两个以上发生时导致的。根据这种情况，可将图 6-3 改造为如图 6-4 所示的等效软件故障树。

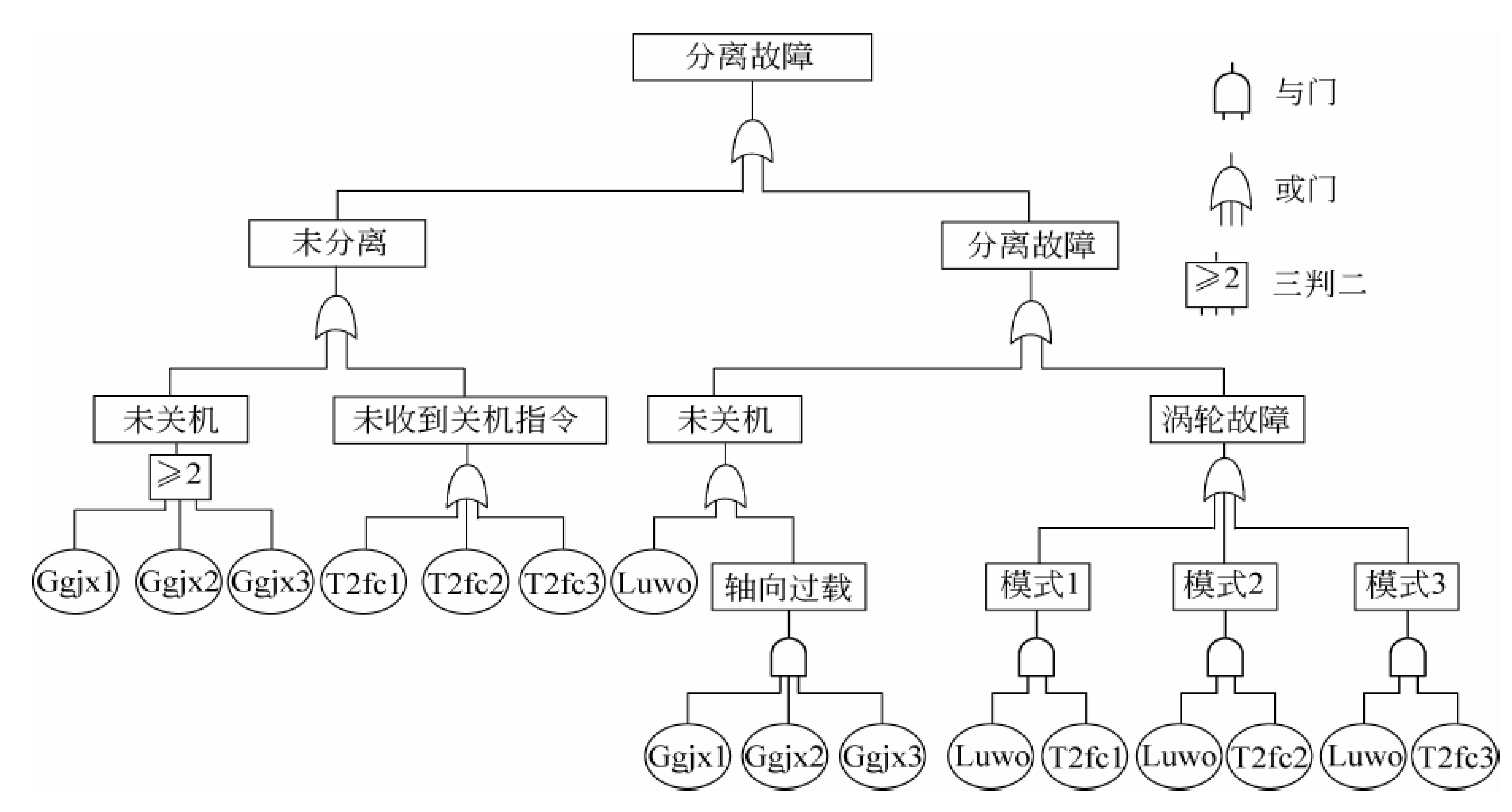



图 6-3　软件故障树

(2) 求出割集为{T2fc1，T2fc2，T2fc3}，{Ggjx1，Ggjx2}，{Ggjx1，Ggjx3}，{Ggjx2，Ggjx3}，{Luwo}，{Ggjx1，Ggjx2，Ggjx3}，{Luwo，T2fc1}，{Luwo，T2fc2}，{Luwo，T2fc3}。

(3) 求出最小割集为{Ggjx1，Ggjx2}，{Ggjx1，Ggjx3}，{Ggjx2，Ggjx3}，{Luwo}，{T2fc1，T2fc2，T2fc3}。

(4) 根据输入条件的等价类划分原则，设置输入的典型值如表 6-6 所示。

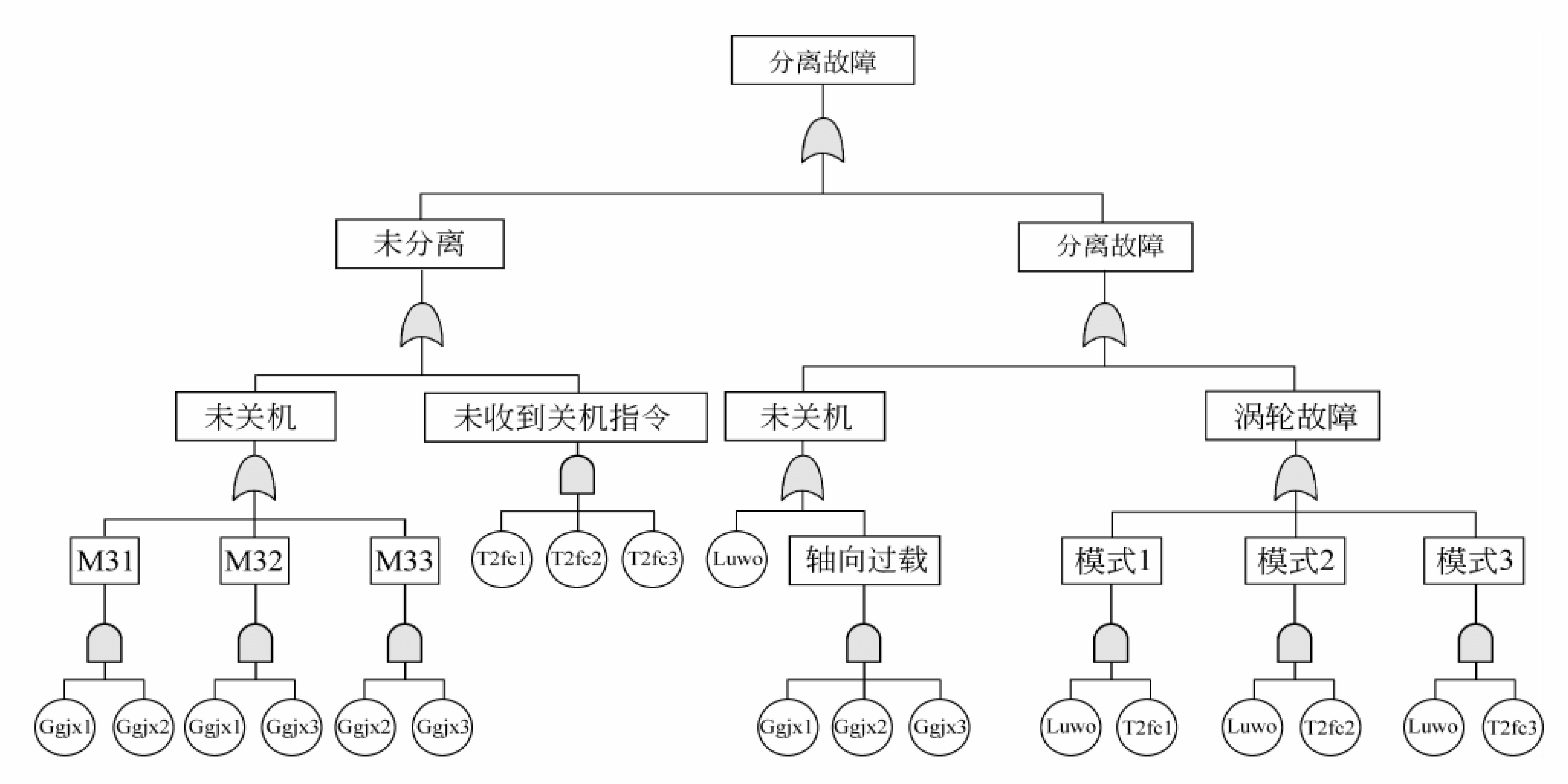



图 6-4 改造后的软件故障树

表 6-6 典型值设计

| 参数名 | 典　型　值 |
|---|---|
| Ggjx1 | 有效值，无效值 |
| Ggjx2 | 有效值，无效值 |
| Ggjx3 | 有效值，无效值 |
| T2fc1 | 有效值，无效值 |
| T2fc2 | 有效值，无效值 |
| T2fc3 | 有效值，无效值 |
| Luwo | 有效值，上越界，下越界，上边界，下边界 |

（5）生成测试用例。基于割集生成的测试用例有 63 个，而基于最小割集生成的测试用例只有 25 个。利用测试用例自动生成工具生成的基于最小割集{T2fc1，T2fc2，T2fc3}的测试用例集合如图 6-5 所示。

| | | |
|---|---|---|
| 18 | T2fc1、T2fc2、T2fc3 | 有效值(T2fc1)、有效值(T2fc2)、有效值(T2fc3) |
| 19 | | 有效值(T2fc1)、有效值(T2fc2)、无效值(T2fc3) |
| 20 | | 有效值(T2fc1)、无效值(T2fc2)、有效值(T2fc3) |
| 21 | | 有效值(T2fc1)、无效值(T2fc2)、无效值(T2fc3) |
| 22 | | 无效值(T2fc1)、有效值(T2fc2)、有效值(T2fc3) |
| 23 | | 无效值(T2fc1)、有效值(T2fc2)、无效值(T2fc3) |
| 24 | | 无效值(T2fc1)、无效值(T2fc2)、有效值(T2fc3) |
| 25 | | 无效值(T2fc1)、无效值(T2fc2)、无效值(T2fc3) |

图 6-5 基于最小割集生成的测试用例示例

## 6.8　配置项测试过程

软件配置项测试依据软件需求文档，按照如图 6-6 所示过程开展测试，配置项测试过程分为 4 个阶段。

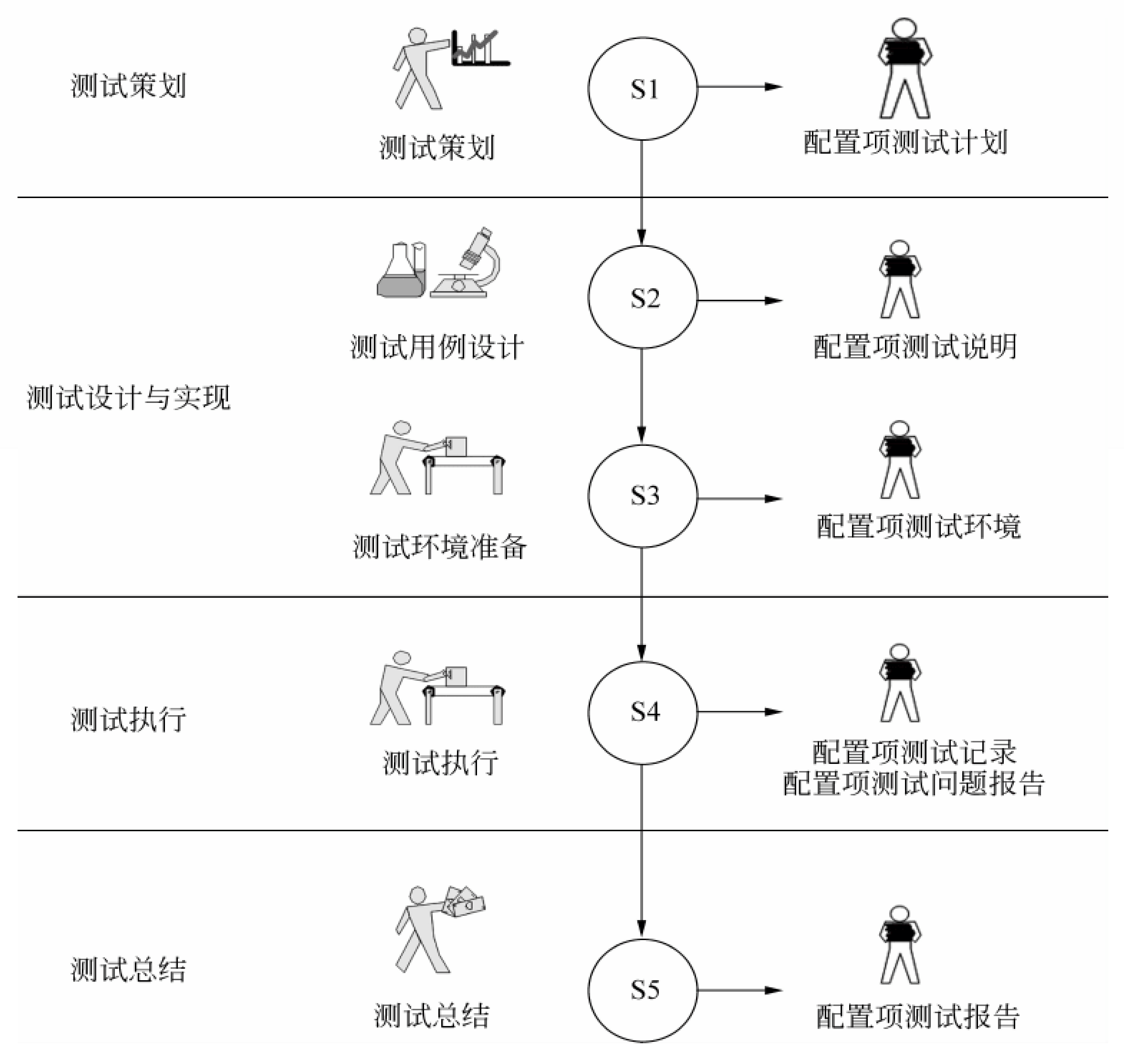



图 6-6　软件配置项测试过程示意图

(1) 配置项测试策划。依据软件需求文档进行配置项测试策划，软件需求文档包括软件研制任务书、软件需求规格说明、接口需求规格说明和软件用户手册，制定软件配置项测试计划。

(2) 配置项测试设计与实现。依据软件需求文档、软件配置项测试计划设计配置项测试用例，开发必要的配置项测试程序。

(3) 配置项测试执行。按照软件配置项测试计划、测试说明明确的测试策略、测试环境，执行测试用例，记录实测结果和测试过程出现的问题。当软件问题修改后还需要进行回归测试。

(4) 配置项测试总结。根据实测结果、期望结果和评估准则分析测试数据，形成测试报告。测试报告完成后，应对整个配置项测试的情况、文档等进行评审，以确保配置项测试的有效性。

### 6.8.1 测试策划

为了保证软件研制进度，一般情况下，可在软件需求分析完成后即开展软件配置项测试策划、测试设计与实现工作。完成配置项测试策划工作时，应编写软件配置项测试计划。

目前，软件研发项目在配置项测试方面还存在着许多问题。主要表现在以下 3 个方面：

（1）对配置项测试的重要性认识不足；

（2）配置项测试开始较晚，大部分项目受资源限制，在整个项目编码和单元测试结束后才开始进行配置项测试，没有根据配置项测试策略，分步骤、分阶段地进行配置项测试工作；

（3）配置项测试用例设计不够全面，大部分问题主要表现在，只关注了配置项的软件功能，未充分覆盖软件接口，另外，很少考虑异常情况的测试。

因此，为了达到质量、进度、效率的平衡，应在配置项测试策划时考虑如下因素：

（1）软件的质量要求；

（2）软件研制的进度安排；

（3）软件测试的优先级；

（4）测试资源的限制。

配置项测试策划的内容如下。

（1）按照软件需求和软件质量要求确定软件配置项测试的需求。包括：

① 被测软件名称、标识；

② 采取的配置项测试策略；

③ 明确需要进行的测试类型；

④ 明确每个测试类型下的测试项；

⑤ 提出每个配置项的测试方法；

⑥ 提出配置项测试的充分性要求；

⑦ 测试终止条件；

⑧ 优先级；

⑨ 对软件需求文档的追踪关系。

（2）提出配置项测试环境要求，包括测试所需的测试程序等。

（3）提出配置项测试人员安排。

（4）安排配置项测试的进度计划。应依据测试需求、测试环境、人员等情况，制定合理可行的软件配置项测试进度计划。

（5）制定配置项测试通过的准则。配置项测试通过的准则如下：

① 配置项的软件功能、性能、接口等需求与软件需求一致；

② 配置项测试发现的问题得到修改并通过回归测试；

③ 满足配置项测试终止条件；

④ 配置项测试报告通过评审。

配置项测试策划完成时，应编写软件配置项测试计划。配置项测试计划是否合理可行、是否满足充分性要求等需要进行验证，因此，应对软件配置项测试计划进行评审。评审的主要内容包括：

(1) 测试人员组成合理、分工明确；
(2) 配置项测试策略恰当；
(3) 配置项测试方法可行；
(4) 测试通过的准则满足软件质量要求；
(5) 测试类型、测试项覆盖软件需求文档和其他软件质量要求；
(6) 测试资源满足配置项测试需求；
(7) 建立软件配置项测试项(条目)到软件需求文档之间的追踪关系。

## 6.8.2 测试设计与实现

配置项测试设计与实现阶段的主要工作是依据软件需求文档和配置项测试计划，进行配置项测试用例的设计，并完成必要的配置项测试环境的建立和验证。配置项测试设计与实现阶段完成时，应编写配置项测试说明、配置项测试所需测试环境和测试环境验证情况报告。

**1. 设计配置项测试用例**

在进行配置项测试用例设计时，应遵循如下原则：
(1) 应针对软件配置项测试计划中提出的测试项进行测试用例设计；
(2) 测试软件需求覆盖率应达到 100%，如果存在无法覆盖的情况，应说明原因；
(3) 测试用例的输入应至少包括有效等价类值、无效等价类值和边界值；
(4) 测试步骤应明确、具体；
(5) 测试数据应说明数据的主要特征；
(6) 应明确测试评估准则，必要时说明误差范围。

**2. 建立配置项测试环境**

配置项测试时，往往会遇到目标运行环境还不完备的情况。因此，建立配置项测试环境时，需要考虑这些风险，尽早制定缓解措施和应急计划，以保证配置项测试工作的顺利实施。

在条件允许的情况下，应尽可能使用目标运行环境以保证配置项测试的有效性。如果确实无法使用真实环境，建立的仿真环境需要经过认真分析和确认，保证与真实环境的等效性。

**3. 测试就绪评审**

为了保证配置项测试的充分性，需要对配置项测试计划、说明和测试环境就绪情况进行评审，评审的内容包括：
(1) 配置项测试策略是否恰当；
(2) 测试类型和测试项是否充分；
(3) 测试项是否包括了测试终止要求；
(4) 是否充分考虑了配置项测试可能的风险，缓解和应急计划是否恰当；
(5) 测试说明是否完整、正确和规范；
(6) 测试设计是否完整和合理；
(7) 测试用例是否可行和充分；

(8) 通过比较测试环境与软件真实运行的软件和硬件环境的差异，审查测试环境要求是否正确合理、满足测试要求；

(9) 文档是否符合规定的要求。

### 6.8.3 测试执行

配置项测试执行活动是依据配置项测试计划与测试说明，按测试执行顺序进行配置项测试，记录实际的测试结果。当发现问题时，应进行分析，如果是测试的问题，应根据实际情况调整测试计划和说明，补充相应的测试；如果是软件的问题，应如实、详细地记录测试结果，并提交问题报告。问题报告中应详细描述问题现象，问题类型和级别，给出改进意见及建议，为随后的问题定位和解决提供支持。

如果软件进行了更改，应进行相应的回归测试。回归测试的详细内容参见第 8 章的介绍。

### 6.8.4 测试总结

软件配置项测试完成后，应对配置项测试工作进行总结，以便评估软件配置项测试中的问题是否得到解决，配置项测试工作是否到达充分性要求。测试结果分析总结应写入软件测试报告，并应进行测试报告评审。

**1. 配置项测试分析总结**

配置项测试分析总结的内容如下。

(1) 对配置项测试过程进行总结。应对配置项测试策划、测试设计与实现、测试执行过程进行总结，说明在各过程中开展的主要工作、参与人员和工作完成情况。

(2) 对配置项测试方法进行说明。应说明配置项测试所采用的测试方法与策略，并说明采用这些方法的依据。

(3) 对配置项测试环境进行分析。应说明配置项测试所使用的测试环境，包括测试工具、测试程序的情况，并对测试环境的差异性进行分析，说明测试环境是否满足配置项测试的要求。

(4) 对配置项测试结果进行分析。配置项测试结果的分析，应包括对测试执行过程以及所有回归测试情况的分析。主要内容包括：

① 测试时间；

② 测试人员；

③ 测试用例执行情况，内容包括测试用例数、通过的测试用例、未通过的测试用例、完全执行的测试用例、未完全执行的测试用例、未执行的测试用例；

④ 测试覆盖情况，包括对功能、性能、接口等需求覆盖情况，说明是否满足测试充分性要求；

⑤ 说明配置项测试过程中发现的问题，并对问题解决情况进行说明；

⑥ 对软件的整体情况进行评价，并提出改进的意见及建议；

⑦ 如果软件中存在遗留问题应进行说明。

**2. 配置项测试总结评审**

配置项测试总结评审的内容包括：

（1）审查测试文档与记录内容的完整性、正确性和规范性；

（2）审查测试活动的有效性；

（3）审查测试环境是否符合测试要求；

（4）审查软件测试报告与软件测试原始记录和问题报告的一致性；

（5）审查实际测试过程与测试计划和测试说明的一致性；

（6）审查测试说明评审的有效性，例如是否评审了测试项选择的完整性和合理性、测试用例的可行性和充分性；

（7）审查测试结果的真实性和准确性；

（8）审查对遗留问题是否有有效的处理措施。

# 第 7 章 CHAPTER 7

# 系 统 测 试

系统测试是对完成集成的系统,包括软件和硬件,在实际运行环境下作为一个整体进行的测试。系统测试是软件产品交付前进行的最后确认,确保交付的软件产品满足用户的需求。系统测试根据系统的研制任务书、系统需求规格说明、用户手册等进行测试,确保系统满足用户的要求,符合系统规格说明中定义的各项需求。同时,系统研制任务书、系统需求规格说明、软件实体、软件用户手册应文文相符、文实一致。

## 7.1 概述

### 7.1.1 系统测试的定义

系统测试的对象是通过集成的整个系统。

系统测试一般情况下,应依据系统研制要求、系统/子系统规格说明、系统/子系统设计说明、接口需求规格说明和用户手册开展功能、性能、接口、强度、容量、余量、可靠性、安全性、安装和人机交互界面测试等。必要时,可进行包括文档审查,以及恢复性、互操作性和兼容性测试等动态测试。

### 7.1.2 系统测试的目的

系统测试的目的主要包括:

(1) 确认系统是否达到了研制任务书、系统/子系统规格说明、系统/子系统设计说明、接口需求规格说明等所规定的各项要求;

(2) 是否满足系统验收交付的要求。

软件研制、测试人员应根据研制任务书、系统/子系统规格说明、系统/子系统设计说明、接口需求规格说明等定义的全部需求制定系统测试计划,开展系统测试工作。

系统测试与软件配置项测试的区别,主要表现在以下 3 个方面。

(1) 测试对象不同。软件配置项测试的对象是软件配置项,而系统测试的对象是整个系统,包括所有软件和硬件。

(2) 测试的依据不同。软件配置项测试的依据是软件配置项的研制任务书和(或)需求规格说明、接口需求规格说明和用户手册,而系统测试的依据是系统的研制任务书、系统/子

系统规格说明、系统/子系统设计说明、接口需求规格说明等。

（3）测试内容存在差异。软件配置项测试需要考察软件研制任务书和（或）软件需求规格说明中描述的功能、性能、接口（系统外部接口）、安全性、恢复性和安装性等是否符合要求。系统测试则关注整个系统是否能够协调一致地完成系统规定的功能、性能，是否正确实现与其他系统间的接口，系统规定的安全性、恢复性、安装性、可靠性、互操作性和兼容性等是否符合要求。

### 7.1.3　系统测试的重要性

系统测试是软件与系统完整结合后的测试，是产品交付前最后的测试，也是最为重要的测试活动之一，其重要性主要表现在以下 5 个方面。

（1）软件第一次完整地与硬件相结合开展的测试。

（2）通过系统测试可以发现软件是否遗漏了系统研制任务书、系统/子系统规格说明、系统/子系统设计说明、接口需求规格说明等需求。

（3）一般情况下，用户将参与系统测试活动，例如：测试就绪评审、测试执行等，使软件完成的功能、性能等得到用户的确认，保证软件交付的顺利进行。

（4）系统测试关注的是整个系统是否满足用户的要求。系统测试可以发现与其他系统的接口、用户使用界面、性能、安全性、安装性等与用户直接相关的问题，避免在交付后系统出现较大的问题。

（5）系统测试在真实环境下开展，对整个系统完成的需求进行考察，可以有效避免在交付后出现系统的不协调问题。

## 7.2　系统测试原则

为保证系统测试充分和有效，在进行系统测试时，应遵循以下原则：

（1）系统测试的依据是系统研制要求、系统/子系统规格说明、系统/子系统设计说明、接口需求规格说明等规定的功能、性能、接口、安全保密性等特性；

（2）功能覆盖率、性能和接口覆盖率应达到 100%；

（3）系统的每个特性一般应被一个正常测试用例和一个异常测试用例覆盖；

（4）测试用例的输入一般应覆盖有效值、无效值和边界值；

（5）应测试系统的输出及其格式；

（6）应测试运行条件在边界状态和异常状态下，或在人为设定的状态下，系统的功能和性能；

（7）应测试系统的全部存储量、输入/输出通道和处理时间的余量；

（8）应按照系统研制要求、系统/子系统规格说明、系统/子系统设计说明的要求，对系统的功能、性能进行强度测试；

（9）应测试系统中用于提高系统安全性、可靠性的结构、算法以及用于容错、冗余、中断处理等方案；

(10) 对安全关键的系统,应对每一个危险状态和导致危险的可能原因进行针对性测试;

(11) 对有恢复或重置功能需求的系统,应测试其恢复或重置功能和平均恢复时间,应对每一类导致恢复或重置的情况进行测试;

(12) 对不同的实际问题应外加相应的专门测试;

(13) 测试人员应尽可能地与开发人员分离;

(14) 系统测试过程中应有用户或用户代表的参与。

## 7.3 系统测试环境

系统测试一般需要在目标环境下进行。如果有测试内容无法使用实际的运行环境时,可以使用有效的仿真环境进行系统测试。但是,应对仿真环境与实际运行环境的差异进行分析,确保仿真环境与实际运行环境是等效的。另外,系统测试需要使用一些专用工具,甚至需要开发一些数据仿真和数据获取工具才能完成相应的测试。

系统测试环境的建立可以从以下 4 个方面考虑。

(1) 硬件环境。系统测试环境应与实际运行环境一致。建立系统测试环境时,应按照系统研制要求中规定的系统运行硬件环境建立系统测试环境。特别应关注硬件环境的各项指标是否与实际环境一致。

(2) 操作系统环境。由于整个系统中软件可能运行在多个操作系统下,因此在进行系统测试时,需要充分考虑软件运行环境中操作系统的要求。应关注操作系统的版本应与系统研制要求保持一致。

(3) 数据库环境。目前以数据库为基础的信息系统非常普遍,因此在进行系统测试时需要根据系统研制要求中规定的数据库环境要求,建立系统测试环境。这是因为不同的数据库系统、不同数据库系统版本的性能、容量都存在不同,在系统测试时要尤其关注这部分内容。

(4) 网络环境。网络环境是影响系统性能等的关键因素,因此在进行系统测试时,应按照研制要求采用实际的网络环境。

## 7.4 系统测试策略

系统测试一般采用黑盒测试技术,系统测试的依据是系统研制要求、系统/子系统需求规格说明、系统/子系统设计说明、接口需求规格说明和用户手册等。

系统测试的输入一般通过仿真程序或通过界面等注入,输出一般也需要测试程序或显示界面获取,对测试结果的分析,有时需要使用相应的工具进行。

进行系统测试应具备以下基本条件:

(1) 具有系统研制要求、系统/子系统需求规格说明、系统/子系统设计说明、接口需求规格说明和用户手册等;

(2) 已完成系统的集成测试;

（3）系统中所有软件配置项已进入配置管理库进行管理；

（4）系统源代码通过编译或汇编；

（5）系统测试通常需要进行功能测试、性能测试、接口测试、人机交互界面测试、强度测试、余量测试、安全性测试、恢复性测试、边界测试、数据处理测试、安装性测试和容量测试，有时还需要进行可靠性、互操作性和兼容性测试，具体的测试类型需根据系统需求确定。

## 7.5　系统测试内容

系统测试的内容应根据系统研制要求、系统/子系统需求规格说明、系统/子系统设计说明和接口需求规格说明等中的要求确定，一般情况下应包括如下内容：

（1）系统研制要求、系统/子系统需求规格说明中定义的功能、性能和安全保密性需求；

（2）系统/子系统设计说明和接口需求规格说明中定义的系统内部接口和外部接口；

（3）应测试系统运行在边界状态和异常状态下，系统的功能和性能；

（4）应测试系统的全部存储量、输入/输出通道和处理时间的余量；

（5）应按照系统/子系统规格说明和（或）设计说明的要求，对系统的功能、性能进行强度测试；

（6）应测试系统中用于提高系统安全性、可靠性的结构、算法，以及容错、冗余、中断处理等方案；

（7）对安全关键的系统，应在安全性分析的基础上对每一个危险状态和导致危险的可能原因进行针对性测试；

（8）对有恢复或重置功能需求的系统，应测试其恢复或重置功能和平均恢复时间，并且对每一类导致恢复或重置的情况进行测试；

（9）对系统的其他需求进行专门测试。

## 7.6　系统测试方法

系统测试一般采用黑盒测试技术，主要是验证系统是否满足系统研制要求、系统/子系统规格说明、系统/子系统设计说明和接口需求规格说明等要求。系统测试一般要完成功能、性能、接口、强度、容量、余量、可靠性、安全性、安装和人机交互界面测试等。必要时，可进行包括文档审查，以及恢复性、互操作性和兼容性测试等动态测试。其中大部分测试方法在第 6 章中已有说明。本节重点对可靠性、互操作性和兼容性测试进行说明。

### 7.6.1　可靠性测试

可靠性测试是在真实的或仿真的环境中，运用建模、统计、试验、分析和评价等手段对软件可靠性实施的测试。通过可靠性测试可以发现并纠正软件的缺陷，提高软件的可靠性，并评估研制软件是否达到了用户的可靠性要求。

可靠性测试一般采用黑盒测试方法，需要按照运行/操作剖面进行测试用例设计。测试目的是发现影响软件可靠性的缺陷，实现软件可靠性的提高，验证软件是否达到可靠性要求，并估计软件可靠性水平。可靠性测试环境包括实验室仿真环境和现场使用环境。一般情况下，可靠性测试在系统测试中进行。

软件可靠性测试流程如图 7-1 所示，包括可靠性测试需求分析、运行/操作剖面构造、测试用例设计、测试场景设计、测试环境和工具的准备、可靠性测试实施、可靠性分析与预测等。

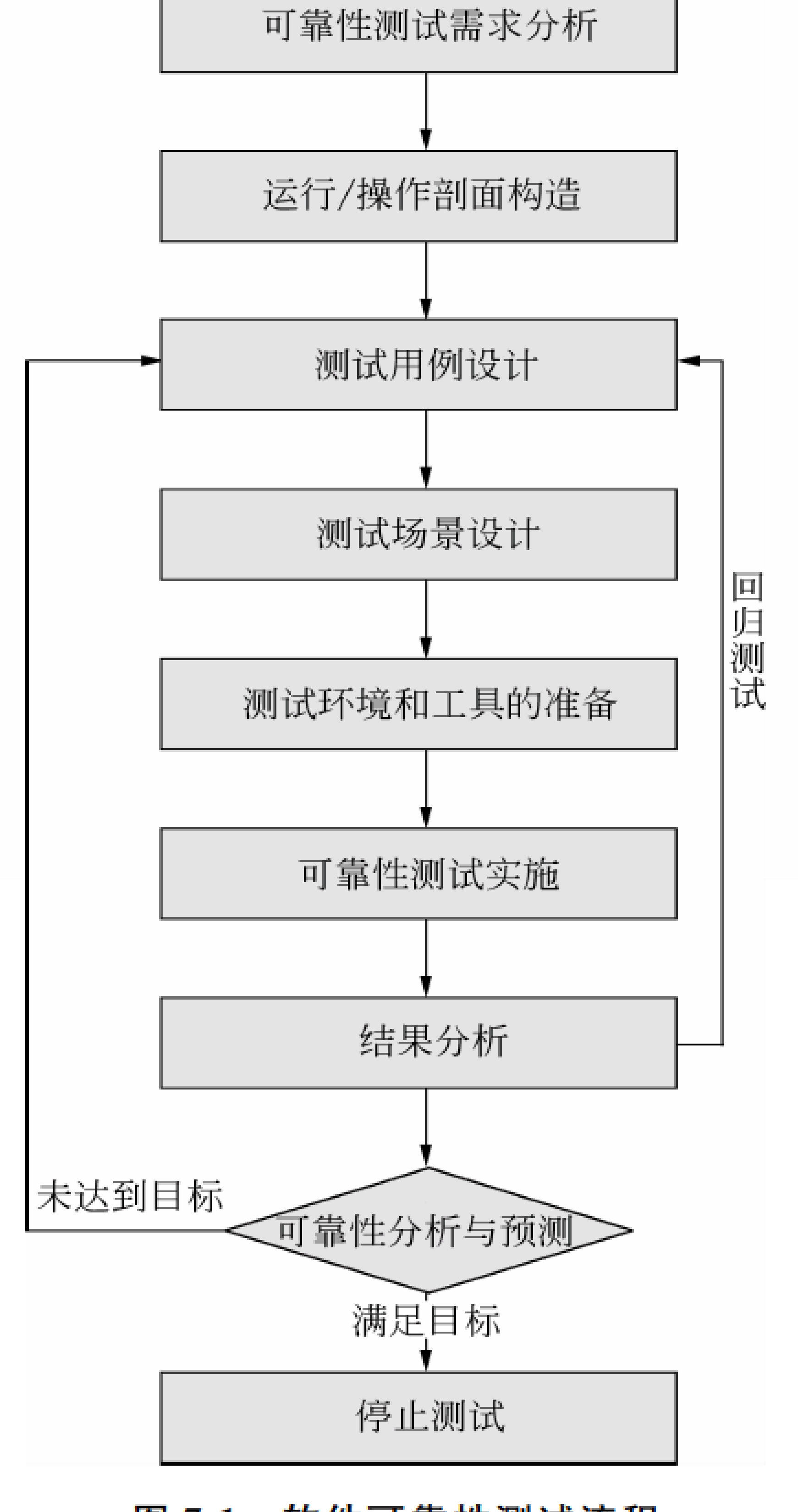


图 7-1 软件可靠性测试流程

**1. 测试技术要求**

可靠性测试的一般要求为：

(1) 可靠性测试环境应与典型使用环境的统计特性相一致，必要时使用测试平台；

(2) 定义软件失效等级，建立软件运行剖面/操作剖面；

(3) 测试记录更为详细、准确，应记录失效现象和时间；

(4) 必须保证输入覆盖，应覆盖重要的输入变量值(所有被测输入值域的概率之和必须大于软件可靠性要求)、各种使用功能、相关输入变量可能组合以及不合法输入域等；

(5) 对于可能导致软件运行方式改变的一些边界条件和环境条件，必须进行有针对性的测试。

**2. 实施要点**

1) 可靠性测试需求分析

可靠性测试需求分析包括定义软件失效等级、明确可靠性测试需求和选择可靠性模型。

(1) 定义软件失效等级。软件失效等级的划分可按照软件失效后造成的影响分为 5 级，失效等级定义见表 7-1。

表 7-1 软件失效等级定义

| 失效等级 | 定义 | 说明 |
|---|---|---|
| 1 级：关键性失效 | 整个系统终止或数据库严重损坏 | 例如：宕机、蓝屏 |
| 2 级：严重性失效 | 重要功能无法正常运行，并且没有替代的运行方式 | 例如：程序错误 |
| 3 级：普通失效 | 绝大部分功能仍然可用，次要功能受到限制或要采用替代方式 | 例如：打印功能错误 |
| 4 级：轻微失效 | 少数功能在有限的操作中受到限制 | 例如：界面不规范 |
| 5 级：可忽略的失效 | 影响未波及最终用户 | |

(2) 明确可靠性测试需求。一般情况下,软件可靠性需求包括:

① 软件能够进行的容错处理,能够防止的误操作和对错误数据进行一定的处理;

② 软件的自动恢复能力;

③ 软件无故障运行时间或无故障操作的次数。

(3) 选择可靠性模型。可靠性模型应尽早确定,在选择可靠性模型时应考虑以下因素:

① 选择比较成熟、应用范围较广的模型作为分析模型,例如马尔科夫模型;

② 模型的输出值应满足软件可靠性需求;

③ 模型需要的数据在软件中应易于收集;

④ 数据的输入能够通过测试工具获取。

2) 运行/操作剖面构造

软件可靠性测试的最突出特点是按照用户实际使用软件的方式来进行测试,软件的运行/操作剖面是定量描述用户实际使用软件的方法。该方法需要充分分析用户使用软件各种模式和功能,以及相应的输入,还需要分析用户使用软件这些模式和功能的概率。这些信息的获取需要对系统研制要求、系统/子系统需求规格说明、系统/子系统设计说明和接口需求文档等进行充分分析。因此,需要测试人员与软件开发人员、用户进行充分、有效地沟通,搜集系统实际使用的历史数据信息并进行分析。系统工作模式和功能划分越完整,概率越准确,建立的运行/操作剖面就越符合实际情况,基于此进行的可靠性测试,得出的可靠性评估和预测的结果就越有价值。软件运行/操作剖面图可按照以下 3 个步骤获得:

(1) 建立用户使用方式与功能之间的关系图,如图 7-2 所示。

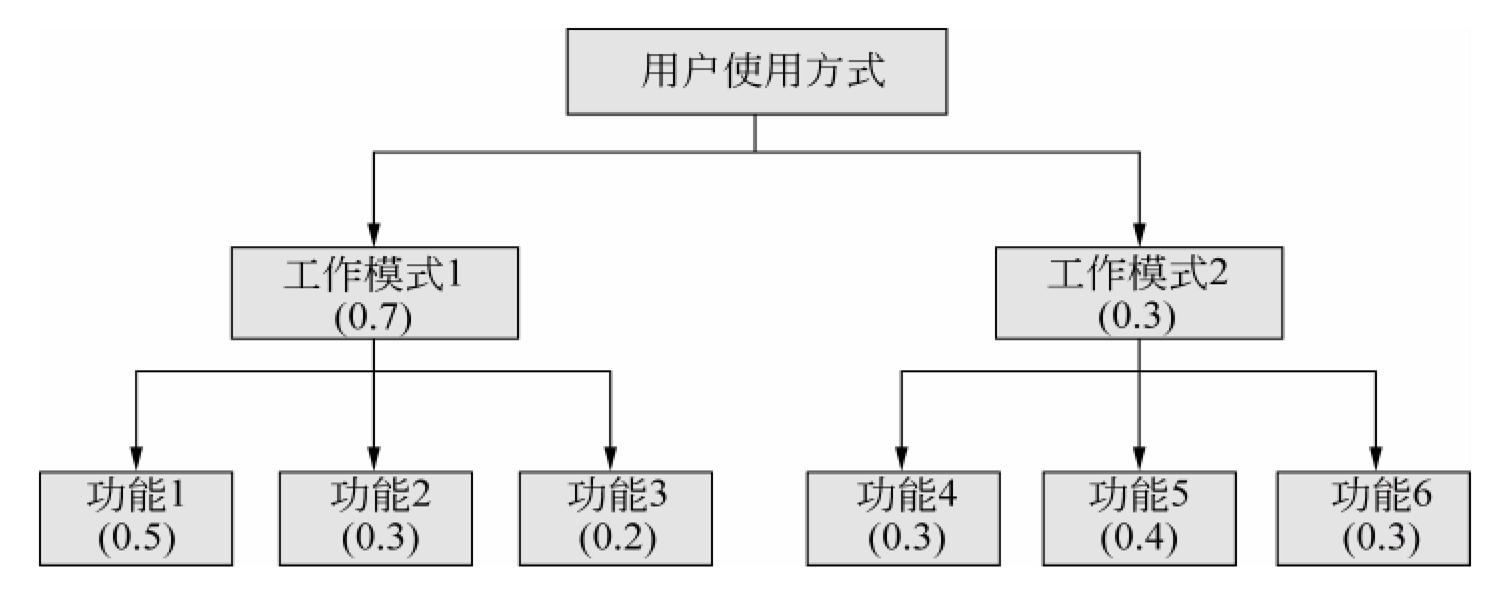



**图 7-2 用户使用方式与功能示意图**

(2) 建立功能与操作之间的关系图,如图 7-3 所示。

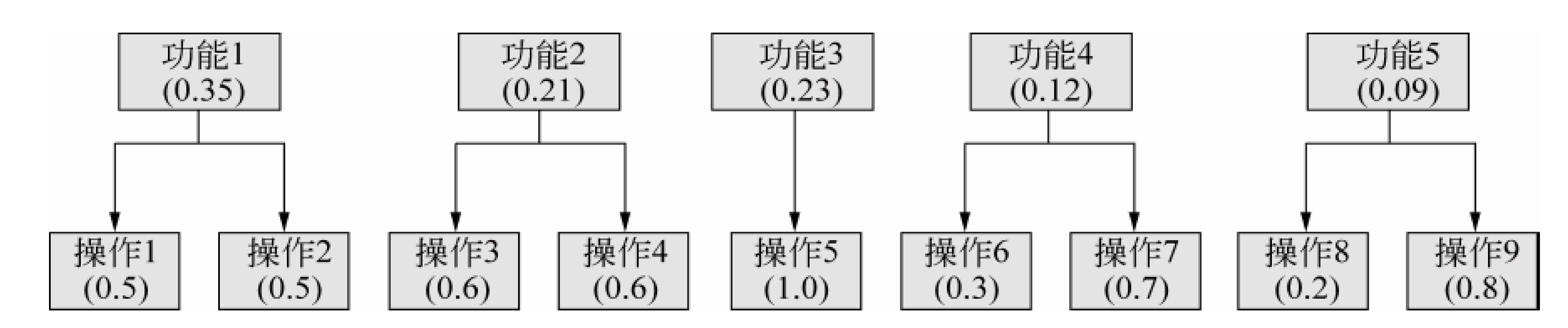



**图 7-3 功能与操作之间关系示意图**

(3) 获得运行/操作剖面,如图 7-4 所示。

3) 可靠性测试用例设计

可靠性测试用例设计时,可根据软件运行/操作剖面,确定功能使用的频率来进行有重

| 操作1 (0.175) | 操作2 (0.175) | 操作3 (0.126) | 操作4 (0.084) | 操作5 (0.230) | 操作6 (0.036) | 操作7 (0.084) | 操作8 (0.018) | 操作9 (0.072) |
|---|---|---|---|---|---|---|---|---|

**图 7-4 获取可靠性测试运行/操作剖面**

点的测试，更真实地反映软件实际使用中的情况，使软件得到更加充分地测试。可靠性测试用例设计时，需要根据对输入数据范围，以及输入数据之间的相互关系的分析来进行设计。另外，应对功能较为复杂的模块设计更多的测试用例，以保证测试的充分性。

4）测试场景设计

测试场景设计可根据如下步骤进行：

(1) 分析软件失效模式，确定关键的失效模式；

(2) 确定软件实际应用中的业务流程；

(3) 估算同时运行软件的用户数量；

(4) 设计用户启动或退出软件的时间；

(5) 设计测试执行周期，确定可靠性测试执行周期和测试时间。

5）测试环境和工具

测试环境和工具的准备主要包括：

(1) 软件运行所需的软/硬件环境和网络环境；

(2) 测试工具所需的软/硬件环境和网络环境；

(3) 测试场景所需的软/硬件环境和网络环境；

(4) 测试数据。

6）测试实施

可靠性测试实施与其他测试类型基本相同，注意需要根据可靠性数据规范和记录方法记录测试数据，例如时间、失效、失效等级等信息。

7）数据分析

可靠性数据分析主要包括失效分析和可靠性分析。

(1) 失效分析时，根据运行结果判断软件是否失效，以及失效的程度、后果、原因等。通过失效分析，找到并纠正引起失效的故障，实现软件可靠性的增长。

(2) 可靠性分析主要是指根据失效数据，估计软件的可靠性水平，预计可能达到的水平，评价软件是否达到要求的可靠性水平。可靠性分析包括：失效密度、失效解决率、故障密度、潜在故障率、故障排除率、测试覆盖率、测试通过率、平均失效间隔时间、有效服务时间率、累计有效服务时间、避免宕机率、避免失效率、防误操作率、平均宕机时间、平均恢复时间、易修复性、修复有效性等，具体的分析方法可参见 GB/T 29832.3—2013《系统与软件可靠性 第 3 部分：测试方法》。

## 7.6.2 互操作性测试

互操作性测试是为验证不同软件之间的互操作能力而进行的测试。

**1. 测试技术要求**

互操作性测试一般针对以下情况进行测试：

（1）同时运行两个或多个不同的软件；

（2）软件之间发生了互操作。

**2. 实施方法要点**

（1）互操作测试时，须同时运行两个或多个不同的软件，且软件之间进行了交互操作。例如：在某系统中，应用程序 S1 初始化时，应用程序 S2 发“检查好”消息，S1 收到后向 S2 发“读取数据 1”命令，S2 向 S1 发送当前数据 1，S1 将 S2 发送的数据 1 进行处理，并向 S2 发送“读取数据 2”命令，S2 向 S1 发送当前数据 2，S1 将 S2 发送的数据 2 进行处理，并完成软件初始化。

（2）应对正常的互操作流程进行测试。

（3）应对互操作流程中可能出现的异常情况进行测试，例如应对接口格式错误、数据异常、流程异常等进行测试。

## 7.6.3　兼容性测试

兼容性测试主要是验证被测软件不同版本之间的兼容性。有两类基本的兼容性测试：向下兼容和交错兼容。向下兼容是测试软件新版本保留它早期版本的功能的情况，交错兼容测试是要验证共同存在的两个相关但不同的产品之间的兼容性。验证软件在规定条件下共同使用若干个实体，或实现数据格式转换时，能满足相关要求的测试。

兼容性测试验证被测软件与其他软/硬件相互是否能够正确交互和实现信息共享。这种交互可能是跨平台的，通过网络进行异地交互，例如微信、QQ 等。兼容性测试需要考虑被测软件与硬件的兼容性、与其他软件平台和应用程序的兼容性以及数据共享的兼容性。

**1. 测试技术要求**

兼容性测试一般需要验证：

（1）验证软件在规定条件下，共同使用若干个实体时满足有关要求的能力；

（2）验证软件在规定条件下，与若干个实体实现数据格式转换时能满足有关要求的能力。

**2. 实施要点**

（1）与硬件的兼容性测试。这项测试内容主要针对通用软件，被测软件在研制过程中应考虑与各类硬件的兼容性。测试时主要从以下 3 个方面考虑：

① 硬件类型方面，例如如果是网络应用软件，应进行多种网络配置的测试，如果是图形应用软件，应进行各种显示器、显卡和声卡的测试；

② 硬件型号和驱动程序方面，针对每种硬件类型应选择主流型号进行测试，例如对于驱动程序，应对操作系统自带的驱动程序、硬件自带的驱动程序，以及官方网站提供的驱动程序进行测试；

③ 硬件特性、模式和选项方面，应对每种硬件的模式和选项进行测试，应考虑最低配置和推荐配置选项。

（2）与其他软件平台和应用程序的兼容性测试。这类测试可从以下方面进行：

① 向前和向后兼容：向前兼容则是指被测软件应与其后续版本保持兼容性，但一般软

件很难做到，这需要被测软件预留相应的接口；向后兼容指被测软件与其以前版本的兼容性，例如采用旧版本保存的数据，在新版本中应该能够正常使用。

② 多个应用程序的兼容性测试：一般情况下，软件不可能独立存在，因此需要对与其他应用程序的兼容性进行测试，可根据软件的种类、使用频度等因素，选择与被测软件交互比较密切、重要的应用程序进行测试，同时需要考虑对这些应用程序的不同版本兼容性的测试。

(3) 数据共享兼容性测试。在对被测软件进行数据共享方面的测试时，应从以下 4 个方面进行测试：

① 文件应能正常保存和读取；

② 文件应能正确导入和导出；

③ 能支持剪切、复制和粘贴；

④ 支持被测软件不同版本间的数据转换，对转换前后的数据进行比较、分析，确保数据不改变业务需求，能处理数据中相互矛盾的地方，特别应支持数据转换不成功时，应不破坏原有数据。

## 7.7 系统测试用例设计

### 7.7.1 概述

系统测试应根据对系统研制要求、系统/子系统需求规格说明、系统/子系统设计说明和接口需求规格说明的分析结果，确定系统测试类型，系统测试一般应包括功能、性能、接口、强度、容量、余量、可靠性、安全性、安装和人机交互界面测试等。必要时，可包括文档审查，以及恢复性、互操作性和兼容性测试等动态测试。在此基础上，分析系统测试项，设计相应的测试用例。

(1) 在进行系统测试用例设计时，应站在系统层面、用户角度考虑测试用例，可以按照用户使用模式、系统工作模式设计测试用例。

(2) 在进行系统测试接口测试时，不仅要关注内部接口测试，更要关注外部接口的测试。内部接口一般情况下在集成测试、配置项测试时都给予了重点测试，因此系统测试应将重点放在系统外部接口的测试上，特别应对外部接口的异常进行重点测试。

(3) 在对系统的强度、容量、余量、安装等进行测试时，需要在系统实际的运行环境中进行，系统的软/硬件环境都应与实际环境一致。

(4) 在进行系统的安全性测试时，可根据第 6 章中介绍的方法进行安全性分析，并在其基础上进行安全性测试用例设计。

(5) 可靠性测试是较为困难的测试，对高可靠性要求的系统进行可靠性测试时，需要采用专门的可靠性测试方法，可以参考 7.6.1 节介绍的方法。

(6) 对有恢复功能的系统进行恢复性测试时，可参考第 6 章介绍的内容。在设计恢复性测试用例时，应考虑以下 5 点：

① 分析系统潜在的危险和系统恢复机制；

② 针对每一种危险状态设计相应的测试用例，验证系统的恢复机制是否正确；

③ 模式的危险应与实际可能发生的危险一致；

④ 应仔细检查恢复机制是否能够记录系统发生危险前的状态，是否能够从该点恢复系统运行；

⑤ 恢复性测试的例子包括是否能够恢复备份的文件、恢复部分文件或恢复数据到系统危险前的状态，系统恢复后是否能够接续系统发生危险前的状态继续执行，双工切换是否能够正确执行，恢复处理日志方面的能力是否符合要求。

对大规模安全攸关系统，开展系统测试是工程实践中的一个难题。本节结合工程实例，提出了一种基于形式化模型的系统测试用例设计方法。首先，对软件功能、性能、接口等进行抽象，给出系统的形式化定义；其次，提出线索分析的不同策略，识别系统测试需求；然后，开展系统的流程、阶段和场景分析，建立可刻画系统级行为的场景树模型；最后，提出基于场景树模型的测试覆盖准则，给出形式化静态检查和动态测试用例生成方法，生成测试用例集合。另外，结合某基于地理信息的安控判决系统进行实例分析和方法描述，给出了实例验证和分析结果。

## 7.7.2　系统形式化模型

**1. 形式化基础**

大规模安全攸关系统一般由多个软件配置项组成，配置项之间存在数据交换或互操作关系，各配置项相互配合实现系统级的功能，并满足所要求的性能指标和其他质量要求。

**定义 1**(功能 FUNC)　FUNC＝(func_Name，$IO_{in}$，$IO_{out}$，$RS_{in}$，$RS_{out}$)称为软件的功能，其中：func_Name 表示功能名称；$IO_{in}=\{pi_1, pi_2, \cdots, pi_m\}(m\geqslant 0)$表示功能的输入，$IO_{out}=\{po_1, po_2, \cdots, po_n\}(n\geqslant 0)$表示功能的输出，$pi_i$ 表示输入参数，$po_j$ 表示输出参数；$RS_{in}$ 表示输入的约束集，$RS_{out}$ 表示输出的约束集。

**定义 2**(性能 PERF)　PERF＝(perf_Name，{(*c*，cv)})称为软件的性能，其中：perf_Name 表示性能名称；$(c, cv)\in C\times CV$，$C=\{c_1, c_2, \cdots, c_n\}$为表示性能的指标集合，$CV=\{cv_1, cv_2, \cdots, cv_n\}(n\geqslant 0)$为表示性能指标的取值范围集合，$cv_i$ 可表示为[min，max]，min 为下限，max 为上限。

**定义 3**(接口 INTF)　INTF＝(intf_Name，sour，dest，pt，DT)称为软件的接口，其中：intf_Name 表示接口名称；sour 表示接口的源地址；dest 表示接口的目的地址；pt 表示接口的协议类型；$DT=\{dt_1, dt_2, \cdots, dt_n\}(n\geqslant 0)$表示接口的数据元素集合。

**定义 4**(配置 CONF)　CONF＝(conf_Name，{(*a*，av)})称为软件的配置，其中：conf_Name 表示配置名称；$(a, av)\in A\times AV$，$A=\{a_1, a_2, \cdots, a_n\}(n\geqslant 0)$表示配置的属性集合，$AV=\{av_1, av_2, \cdots, av_n\}(n\geqslant 0)$表示配置属性的取值集合。

**定义 5**(软件配置项 CSCI)　CSCI＝(csci_Name，$FUNC^{csci}$，$PERF^{csci}$，$INTF^{csci}$，$CONF^{csci}$)称为软件配置项，其中：csci_Name 表示软件配置项的名称；$FUNC^{csci}$，$PERF^{csci}$，$INTF^{csci}$，$CONF^{csci}$分别表示软件配置项的功能、性能、接口和配置的集合。

**定义 6**(系统 SS)　SS＝(ss_Name，$FUNC^{ss}$，$PERF^{ss}$，$INTF^{ss}$，$CONF^{ss}$，CSCI)，其中：ss_Name 表示系统的名称；$FUNC^{ss}$，$PERF^{ss}$，$INTF^{ss}$，$CONF^{ss}$分别表示系统的功能、性能、接

口和配置的集合，CSCI 表示系统所包含的软件配置项的集合。需要说明的是，系统在包含软件配置项集合的同时，还具有自己独立的功能、性能、接口和配置等。

由此，大规模安全攸关系统的静态结构图可以形式化地表示成如图 7-5 所示的样子。

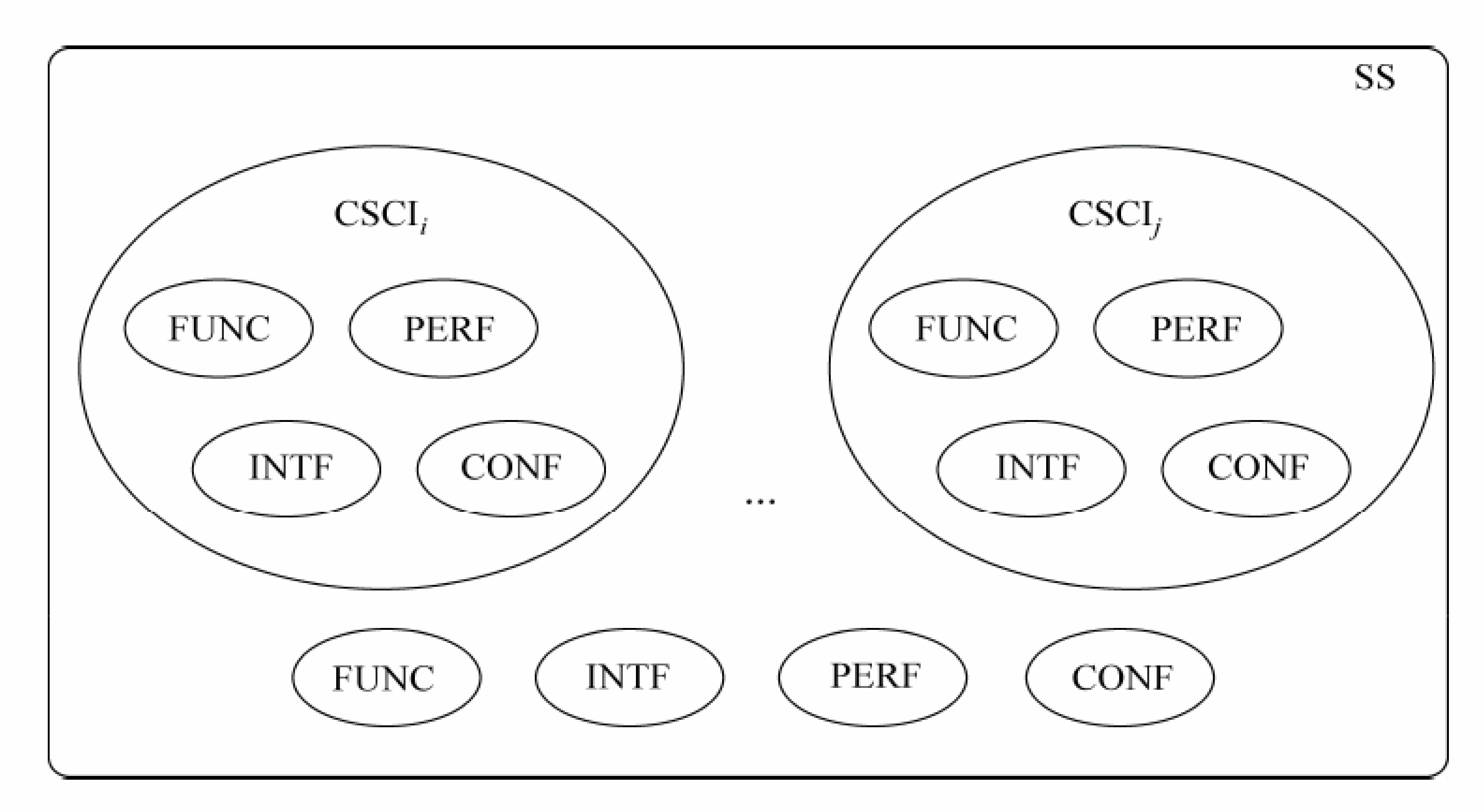



**图 7-5　系统形式化结构图**

**2. 线索分析**

与软件配置项相比，系统级的功能、性能、接口和配置等信息经常是不明显的，特别是在缺少系统级的规格说明时，它们都隐含地分布在各个软件配置项的规格说明或接口文档之中。

系统测试的测试需求分析的难点就在于识别系统级的功能、性能、接口和配置。线索分析方法能够辅助测试人员解决这个问题。

**定义 7**(系统线索 ST)　ST 称为系统线索，指的是在系统层面上可观测的从系统输入到系统输出的映射关系的一个实例。从广义上讲，系统本身就是从输入到输出的一个映射关系，尽管这个映射关系的内容对系统测试人员来讲可能是未知的。

对大规模安全攸关系统开展系统线索分析，主要有以下 4 种策略：

(1) 基于输入进行线索分析。这里的输入包括系统界面或端口的输入数据和用户操作等。设计某种输入的覆盖指标，比如遍历每个界面的各种输入组合(很难实现)或者覆盖常见操作、快捷操作和异常操作序列等，根据指标要求设计系统输入，进而识别系统功能和场景。

(2) 基于输出进行线索分析。这里的输出包括系统界面或端口的输入数据和信息显示等。设计某种输出的覆盖指标，比如遍历每种系统输出，根据指标要求尝试设计相应系统输入，进而识别系统功能和场景。

(3) 基于界面或端口进行线索分析。在每个界面(或端口)检查哪些操作(或数据)会出现，然后根据每个界面(或端口)的操作(或数据)列表查找系统线索。从某种意义上，基于输入的线索分析覆盖了从输入到用户界面的一对多联系，而基于界面的线索分析覆盖了从界面到输入的一对多联系，可看作是对基于输入的线索分析的有益补充。

(4) 基于数据进行线索分析。设计对数据的覆盖指标，然后根据该指标进行线索分析。基于数据进行线索分析主要针对转换式(而不是反应式)系统，这种系统支持以数据库为基

础的事务处理。利用数据及数据之间的关系进行线索分析可产生很多有意思的系统测试线索。

利用线索分析方法能够有效地辅助识别系统级的功能、性能、接口和配置等信息，进而建立场景树。

**3. 场景树模型**

软件系统在某个指定时刻，将处于某个可知的状态，它会调用一个或多个系统功能，并通过系统功能连接一个或多个的系统接口，即开启系统的一个场景。

场景树是本书形式化模型的核心。在场景树之前，首先给出场景和事件的定义。

**定义 8**（场景 A）　$A=(\mathrm{FUNC}_t)$称为软件系统运行中的场景，其中 $\mathrm{FUNC}_t$ 表示系统运行到 $t$ 时刻时可调用的 FUNC 的集合，集合中可包含 0 个、1 个或多个 FUNC。

**定义 9**（事件 E）　$E=(\mathrm{Event_Name},\mathrm{PC},\mathrm{FUNC},\mathrm{PF})$称为触发场景切换的事件，其中：Event_Name 表示事件名称；PC 表示事件的前置条件集合，当前置条件为真时，事件将被触发；FUNC 表示该事件触发的功能集合，PF 为功能对应的参数集合。

系统的执行过程可看作是大量场景的有序序列，从一个场景经由事件触发切换到另一个场景。在场景的切换流程中，由于事件参数的不同，调用不同的功能，进而产生不同的流转方向，因此可以用场景树来描述各场景及其切换关系。在初始场景确定的情况下，系统的执行过程也可看作是一系列事件的有序序列。

**定义 10**（场景树 AT）　$\mathrm{AT}=(A,a_0,E,R_{\mathrm{A,E}})$称为描述各场景及其切换关系的场景树，其中：$A$ 是场景的集合，$a\in A$ 是系统的一个场景，称为场景树的一个节点；$a_0\in A$ 为初始场景；$E$ 为事件的集合，包含触发场景切换的各个事件；$R_{\mathrm{A,E}}\subseteq A\times A\times E$ 表示场景的切换关系集合。如果存在一个切换关系$(a_p,a_c,e)\in R_{\mathrm{A,E}}$使得从场景 $a_p$ 切换到场景 $a_c$，则称 $a_p$ 为 $a_c$ 的父节点，$a_c$ 为 $a_p$ 的子节点，$e$ 为触发该场景切换的事件。

下面以基于地理信息的安控判决系统（简称：地理安控系统）为例，建立其场景树。首先简单介绍地理安控系统的功能和运行流程。

地理安控系统的主要功能是提供地形、地貌、城市、人口、交通枢纽、重要设施等地理信息，显示飞行器飞行过程的测量态势、空间态势、安控态势、姿态变化、运动趋势等试验状态，为方案制定、安控实施、态势掌控等提供决策依据。地理安控系统共包含地理信息存储与管理、试验信息存储与管理、仿真与分析、安控判决处理、实时数据收发和综合态势显示等 6 个软件配置项。

地理安控系统的运行流程是：首先，分别由地理信息存储与管理、试验信息存储与管理配置项提供地理信息、试验信息等系统配置信息，由仿真与分析配置项生成安全管道理论数据，进行系统的初始配置；在配置完成后，系统进入实时处理状态，开始进行实时数据接收和发送；每当接收到实时数据后，由安控判决处理软件进行数据处理，然后把处理结果送综合态势显示配置项显示，同时进行试验信息存储；任务结束后，由仿真与分析配置项开展试验数据事后分析。

根据地理安控系统的运行流程，绘制运行流程图，如图 7-6 所示；然后开展阶段分析，建立地理安控系统阶段分析图，如图 7-7 所示。

结合线索分析技术，基于运行流程图和阶段分析图，绘制地理安控系统的场景树图，如图 7-8 所示。

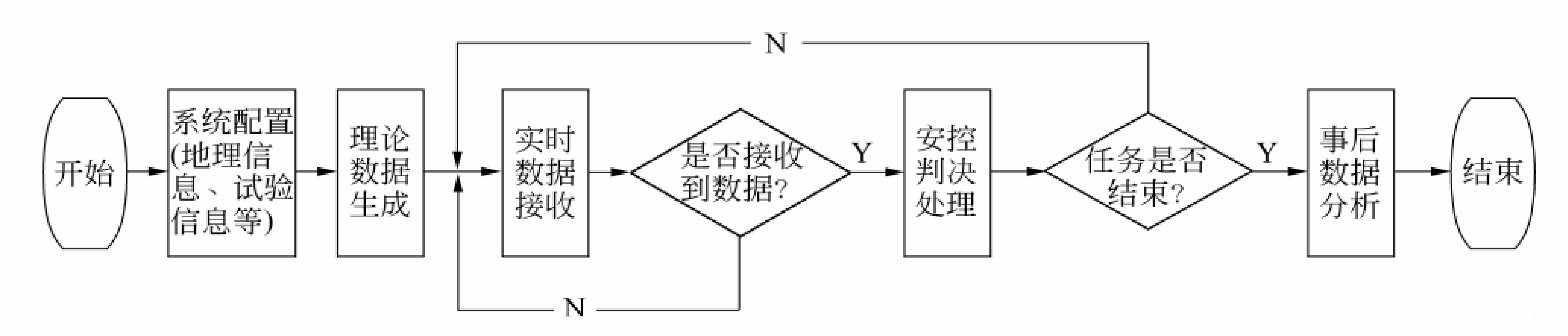



图 7-6 地理安控系统运行流程图

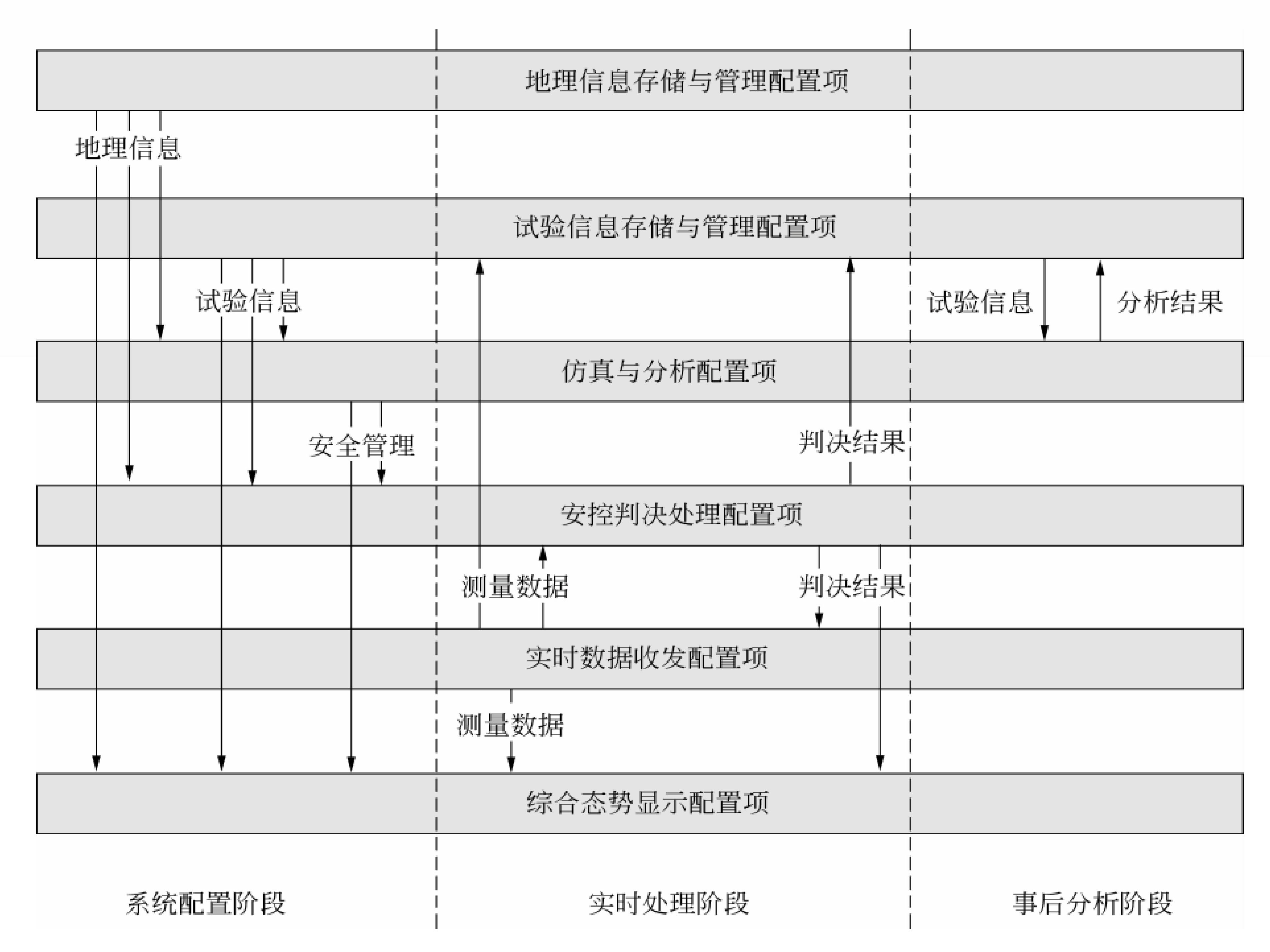



图 7-7 地理安控系统阶段分析图

### 7.7.3 基于模型的系统测试

基于模型的系统测试包括静态配置一致性检查和动态测试。

**1. 静态配置一致性检查**

配置一致性检查是对系统配置信息进行确认的手段，目的是保证当前的配置信息与任务或系统需求是一致的。配置一致性检查一般在动态测试前开展，也可单独开展。

配置一致性检查的做法是把系统的配置信息与给定的标准配置进行比对，检查其是否符合配置一致性的要求。

对于两个配置 $conf_1=(confName_1,\{(a_i,av_i)\})$ 和 $conf_2=(confName_2,\{(a_j,av_j)\})$，其中 $(a_i,av_i)\in A_1\times AV_1$ 和 $(a_j,av_j)\in A_2\times AV_2$，如果同时满足以下 4 个条件，称 $conf_1$ 和 $conf_2$

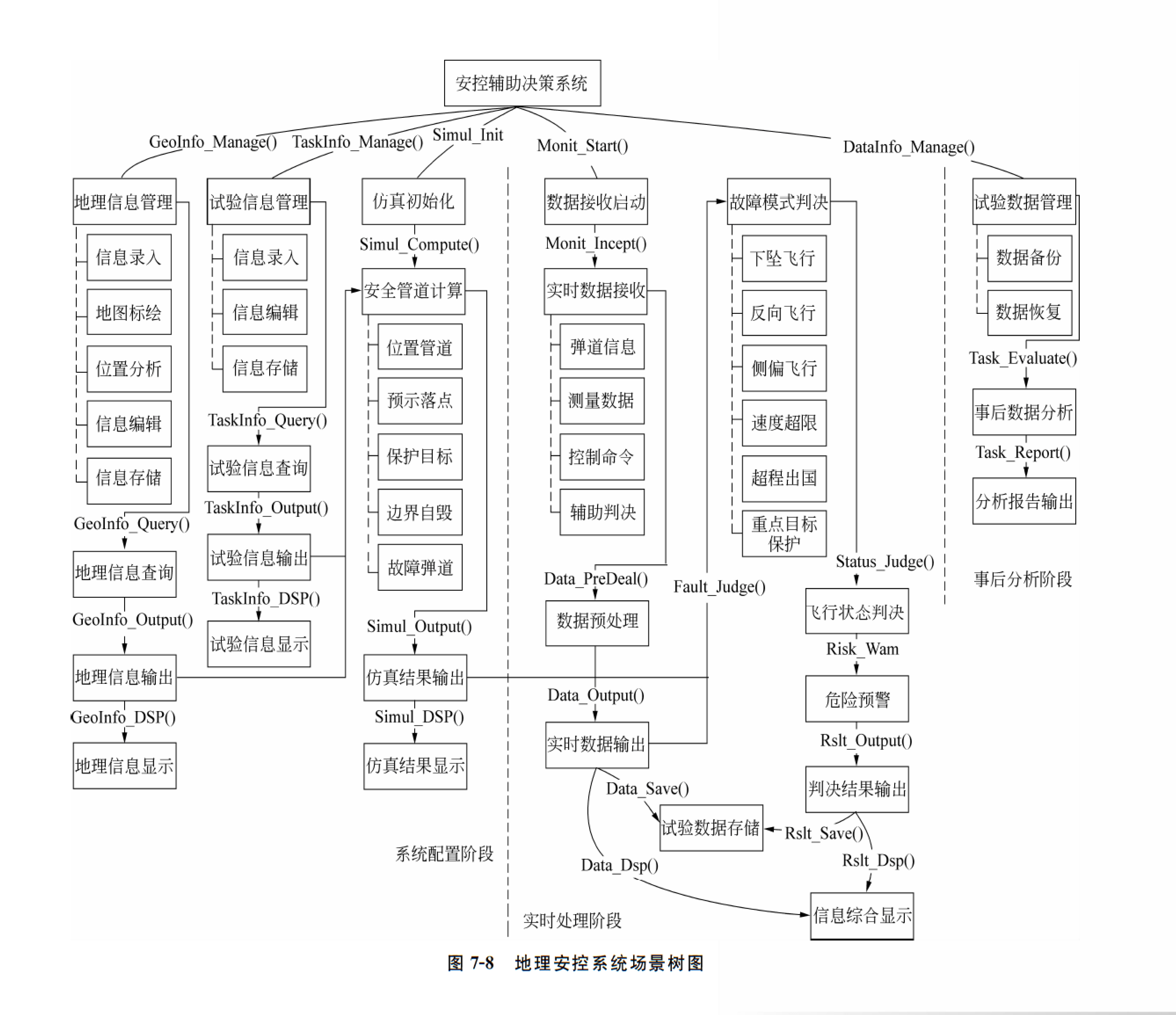



图 7-8 地理安控系统场景树图

是一致的(记为 $conf_1 == conf_2$)：①$confName_1 = confName_2$；②$A_1 = A_2$；③$AV_1 = AV_2$；④对任一$(a_i, av_i) \in A_1 \times AV_1$,存在$(a_j, av_j) \in A_2 \times AV_2$,使得 $a_i = a_j, av_i = av_j$。否则,$conf_1$ 和 $conf_2$ 不一致,记为 $conf_1 != conf_2$。

配置一致性检查的流程如图 7-9 所示,其中标准配置集 ST 是预先存储的包含完整的、标准软件配置信息的配置数据集。一种简单的实现算法如算法 1 所示。

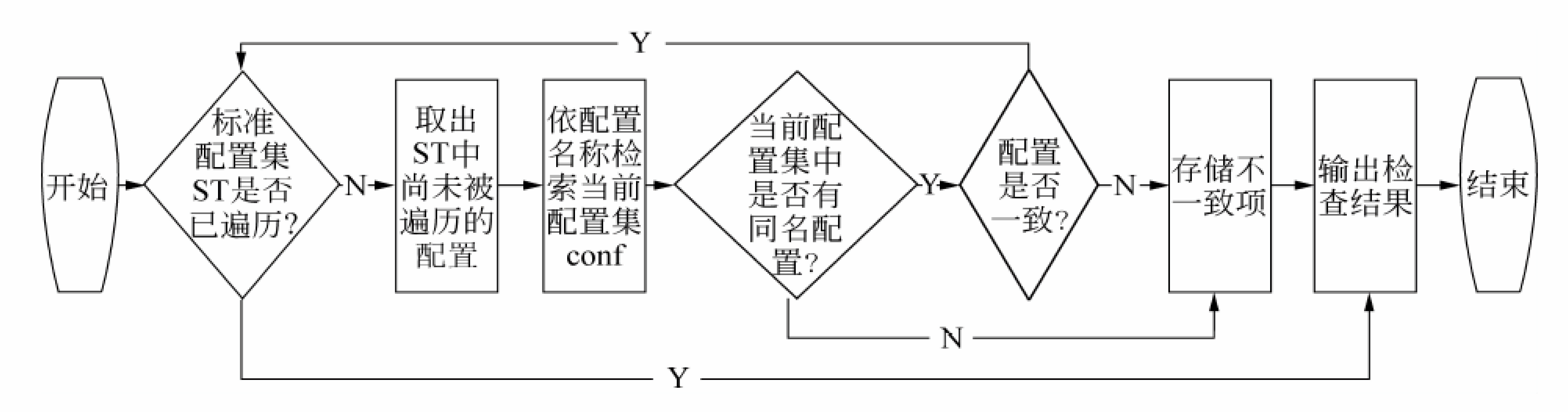



**图 7-9 配置一致性静态检查流程图**

**算法 1：静态配置一致性检查算法**

```
ConfigSetstandardConfSet;
CheckConfigConsistency(ConfigSetcurrentConfSet)
{
ConfigSetconfigCheckRslt;
for(inti=0; i<standardConfSet.count; i++)
{
standardConfSet[i].flag=false;
for(int j=0; j<currentConfSet.count; j++)
{
if(currentConfSet[j] ==standardConfSet[i])
{
standardConfSet[i].flag=true;
}
}
if(standardConfSet[i].flag ==false)
{
configCheckRslt.add(standardConfSet[i]);
}
}
return configCheckRslt;
}
```

配置一致性检查是一种静态测试方法,能够为系统运行提供一个可信的静态环境,保障动态测试结果的有效性与正确性。

**2. 动态测试用例的生成**

参考基于路径的覆盖测试准则,提出基于形式化模型的系统测试的覆盖准则。首先给出覆盖的定义。

**定义 11** 称场景 $a$ 被测试用例集 $T$ 覆盖,如果对 $a$ 包含的任一功能,存在 $T$ 的测试用例子集 $T' \subseteq T$,满足对该功能的测试充分性要求。

**定义 12**　称事件 $e$ 被测试用例集 $T$ 覆盖，如果对 $e$ 触发的任一场景切换，存在 $T$ 的测试用例子集 $T''\subseteq T$，满足对该场景切换情形的测试充分性要求。

对于一个系统 SS，基于其场景树 AT 而生成的测试用例集合 $T$ 应满足以下准则：

(1)（节点覆盖准则）测试用例集 $T$ 满足节点覆盖准则，当且仅当对场景树 AT 中的任一场景（即节点，叶子）$a$，存在测试用例子集 $T'\subseteq T$，使得 $a$ 被 $T'$ 覆盖；

(2)（边覆盖准则）测试用例集 $T$ 满足边覆盖准则，当且仅当对场景树 AT 中任一事件（即边，每两个节点间的树枝）$e$，存在测试用例子集 $T''\subseteq T$，使得 $e$ 被 $T''$ 覆盖。

容易证明，满足节点覆盖和边覆盖的测试用例集就满足如下定义的路径覆盖：测试用例集 $T$ 满足路径覆盖准则，当且仅当对场景树 AT 中任一从根节点（即起始场景）到叶节点的路径（包含其上所有节点和任意两个节点间的边）route，测试用例集 $T$ 能够覆盖 route 上的所有场景和事件。

前面讲过，系统的执行过程是一系列场景的有序序列，也可看作是一系列事件的有序序列，因此采用深度优先遍历方法生成测试用例集，主要思想是：对场景树的任一节点（即场景 $a$），设计覆盖该节点对应路径的一组测试用例，如果该节点为叶节点，则将这一组测试用例添加到测试用例集合中，并设置该节点的状态为已遍历；否则（该节点为父节点），则从该节点未被遍历的子节点中选取一个节点 $a_1$，并从 $a_1$ 出发进行深度优先遍历。基于场景树的动态测试如图 7-10 所示。下面给出测试用例生成算法。

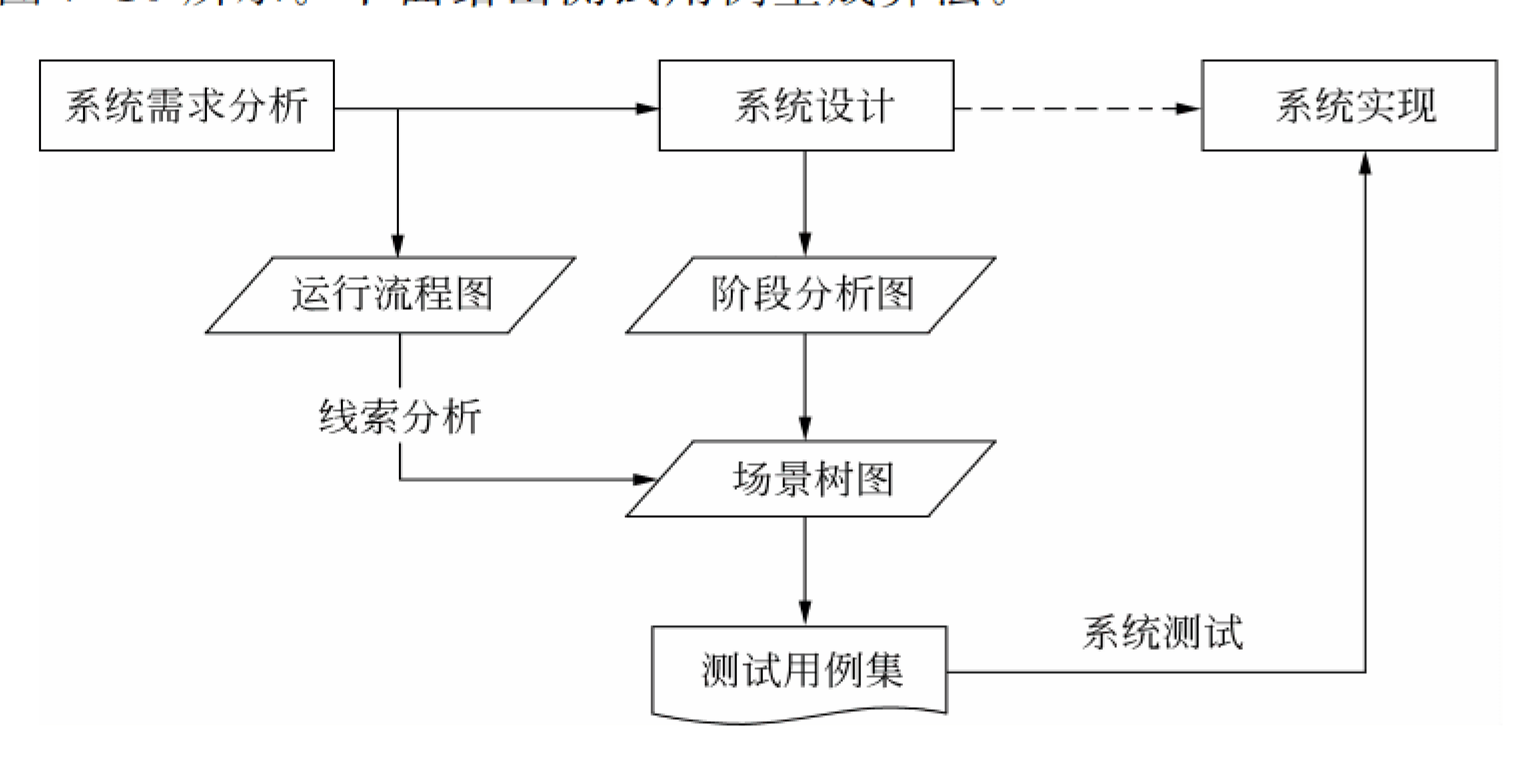



**图 7-10　基于场景树的动态测试**

**算法 2：动态测试用例生成算法**

```
TestCaseSetcaseSet;
CreatTestCaseSet(ActsTreeNode* root, TestCaseArraycaseArray)
{
caseArray.add( GetTestCaseArray(root) );

if(root.countOfChildren ==0)
{
caseSet.add(caseArray);
root.visited =true;
return;
}
```

```
inti =0;
while( (i<root.countOfChildren) && (root.child[i].visited ==false) )
{
CreatTestCaseSet(root.child[i], caseArray);
i++;
}
}
```

### 7.7.4 实例

以地理安控系统为例，对该方法进行了验证。前面已经介绍了地理安控系统的主要功能和作用，绘制了地理安控系统的运行流程图、阶段分析图，并基于运行流程图和阶段分析图，绘制了地理安控系统的场景树图(图 7-8)，这里不再赘述。

基于场景树图，遵循节点覆盖和边覆盖准则，共对地理安控系统设计测试用例 252 个，形成测试用例集合。测试用例的分布如表 7-2 所示。执行测试用例共发现各种软件缺陷 57 个，缺陷分布情况如表 7-3 所示，缺陷分类统计如表 7-4 所示。

**表 7-2 测试用例分布一览表**

| 用例关注点 | 用例数量 | 用例占比 |
|---|---|---|
| 地理信息存储与管理 | 48 | 19.05% |
| 试验信息存储与管理 | 21 | 8.33% |
| 仿真与分析 | 27 | 10.71% |
| 安控判决处理 | 32 | 12.70% |
| 实时数据收发 | 22 | 8.73% |
| 综合态势显示 | 36 | 14.29% |
| 其他(综合流程、系统接口、性能等) | 66 | 26.19% |
| 合计 | 252 | 100% |

**表 7-3 软件缺陷分布一览表**

| 缺陷位置 | 缺陷数量 | 缺陷占比 |
|---|---|---|
| 地理信息存储与管理 | 16 | 28.07% |
| 试验信息存储与管理 | 8 | 14.04% |
| 仿真与分析 | 16 | 28.07% |
| 安控判决处理 | 0 | 0.00% |
| 实时数据收发 | 7 | 12.28% |
| 综合态势显示 | 4 | 7.02% |
| 其他(综合流程、系统接口、性能等) | 6 | 10.53% |
| 合计 | 57 | 100% |

表 7-4 软件缺陷分类统计表

| 严重性等级<br>类别 | 1 级 | 2 级 | 3 级 | 4 级 | 小计 |
|---|---|---|---|---|---|
| 程序错误 | 0 | 0 | 6 | 29 | 35 |
| 文档错误 | 0 | 0 | 0 | 2 | 2 |
| 设计错误 | 0 | 0 | 1 | 15 | 16 |
| 其他错误 | 0 | 0 | 0 | 4 | 4 |
| 小计 | 0 | 0 | 7 | 50 | 57 |

从表 7-2 得出如下结论：

（1）除地理信息存储与管理外，测试用例数量分布情况与地理安控系统各功能部分的比重大致相当。

（2）地理信息存储与管理的测试用例所占比例偏大，是由于存在较多的信息录入、标绘、编辑等人机交互，而根据线索分析的策略，这些人机交互属于测试中应该检查的内容。

（3）其他（综合流程、系统接口、性能等）方面的用例占比接近三成，略低于系统测试的预期，主要是由于地理安控系统相对独立功能占比较大，例如地理信息存储与管理、试验信息存储与管理、仿真与分析部分中的系统配置功能，仿真与分析部分的事后分析功能等。

从表 7-3、表 7-4 可以得出如下结论：

（1）缺陷数目及分布情况与预期相符。地理安控系统的安控判决处理、综合态势显示等功能部分是从已有软件改造而来的，相对成熟、缺陷较少；地理信息存储与管理、仿真与分析等功能部分是新研的，缺陷较多。

（2）其他方面（综合流程、系统接口、性能等）的缺陷较少，是由于地理安控系统各软件配置项通过网络数据帧方式进行信息交换，部分缺陷已反映在实时数据收发功能中。

（3）从缺陷的严重等级看，地理安控系统的缺陷级别普遍较低，没有发现 1 级/2 级的严重缺陷，这是与系统的核心功能部分继承自已有软件相关的。

（4）从缺陷的类别看，程序错误占比超过 60％、设计错误接近 30％，地理安控系统应重点从软件设计的符合性、编码实现的正确性等方面加以改进，这与测试人员对该项目在配置管理、过程控制等方面的直观感受相符。

实例验证结果表明，使用形式化技术能够建立系统的场景树模型，通过所建立的场景树模型能够有效帮助测试人员了解系统行为，识别系统级功能、性能、接口，进而辅助生成满足覆盖准则的系统测试用例集合，测试用例执行结果和软件缺陷分布情况符合预期。地理安控系统在工程实践中的实际效果也印证了该技术是可行的和有效的。

大规模安全攸关系统的软件种类多、规模大，软件间信息交换关系复杂，具有高安全可靠性等特点。利用形式化建模和线索分析方法建立系统的场景树模型，再以场景树模型自动化地辅助生成系统测试的测试用例集的软件测试技术，能够在一定范围内满足对大规模安全攸关系统的测试需求。使用形式化描述和形式化模型开展系统测试具有测试过程清晰、易于确定测试充分性准则、可适用静态检测以及易于自动化实现等优点，同时软件形式化方法在测试需求分析和测试设计中会带来相对较大的测试花销，因此针对系统关键性质或关键业务部分地引入形式化方法是一种有效的测试途径。

## 7.8 系统测试过程

系统测试过程与软件配置项测试过程基本一致，本节在6.8节的基础上简要说明系统测试的过程。

**1. 系统测试策划**

（1）依据系统研制任务书、系统设计说明、用户手册、系统接口需求与设计文档等明确系统测试内容、测试策略，制定系统测试计划。一般初步的系统测试计划在系统设计阶段就可以拟制，随着系统研制的不断完善。

（2）提出系统测试环境的要求，包括测试工具、测试仿真软件、结果获取软件和数据分析软件的需求。

（3）在对系统测试工作量估计的基础上，依据软件研制进度制定系统测试的进度安排。

（4）对系统测试计划进行评审。

**2. 系统测试设计与实现**

（1）依据系统研制任务书、系统设计说明、用户手册、接口需求与设计文档和系统测试计划设计系统测试用例。

（2）获取系统测试工具，开发系统测试仿真软件、结果获取软件和数据分析软件。

（3）对系统测试说明进行评审。

**3. 系统测试执行**

按照系统测试计划、测试说明中明确的测试策略、测试环境，执行测试用例，记录实测结果和测试过程出现的问题。当软件问题修改后还需要进行系统级回归测试。

**4. 系统测试总结**

根据实测结果、期望结果和评估准则分析测试数据，形成测试报告。测试报告完成后，应对整个系统测试的情况、文档等进行评审，以确保系统测试的有效性。

# 回归测试

在软件研制过程中,软件变更不可避免,而变更极有可能引入新的错误。为了确认更动达到预期目的,并且没有引入新的错误,我们就应该对更动后的软件进行回归测试。对于目前使用较为广泛的迭代软件开发模型,更需要不断地进行回归测试。回归测试是有效保证软件质量的重要手段,但由于回归测试不充分而导致失败的例子有很多,阿丽亚娜 5 型火箭发射失败就是由对复用的代码未进行有效的回归测试而造成的。

回归测试可以发生在任何级别的测试过程中,需要根据软件变更的情况采取适合的测试策略和软件更动影响域分析方法,以便有效地进行回归测试用例的设计,保证回归测试的充分性及高效性。

## 8.1 概述

### 8.1.1 回归测试的定义

软件回归测试是软件被修改或扩充(如功能增强或升级)后重新进行的测试,是为了验证修改后的软件是否存在错误而重新进行的测试。在软件开发过程中,软件的变更可能是由于以下 4 种情况引发的。

(1) 纠错性更动。修正在测试等验证确认过程中发现的错误而进行的变更,包括因设计错误、程序错误、数据错误、文档错误而进行的纠错更动。

(2) 适应性更动。为适应软件运行环境改变而做的更动,包括因硬件配置的变化、数据格式或文件结构的改变、软件支持环境的改变而引起的更动。

(3) 完善性更动。为扩充功能或改善性能而进行的更动,包括为扩充和增强功能、改善性能、便于维护而进行的更动。

(4) 预防性更动。为防止问题发生所进行的更动。

由于软件的更动可能存在以下问题。

(1) 更动本身可能不正确。由于对软件缺陷本身分析、定位存在问题造成软件更动本身不正确。根据数据表明,多达一半的缺陷修复在回归测试中没有通过。

(2) 新增代码实现存在错误。对新增功能的代码实现未达到相应要求。

(3) 对软件其他部分造成影响。由于对缺陷更动影响分析不到位,新增的模块或对缺陷部分进行的修改,都可能对原有正确实现的模块造成影响,导致出现新的问题。

因此，在进行回归测试时，不仅需要对变更的部分进行测试，证明其实现了正确的修改，还需要补充新的测试用例对新增代码进行测试。另外，还需要选择部分已有的测试用例来证明对未更动部分没有引入新的错误。

回归测试的对象可以是软件单元、软件配置项、子系统、系统，也可以是集成过程中的中间件。

## 8.1.2 回归测试的目的

软件回归测试是为了检验是否成功修正了缺陷且是否会引起原有正常功能出现新的缺陷而进行的测试。回归测试的目的主要包括以下内容。

（1）验证是否正确地解决了前次测试所发现的问题。为了避免开发人员未真正修复软件问题，应根据软件更改的情况设计相应的测试用例，验证是否正确地修改了存在的问题。

（2）验证软件的变更是否未引入新的问题。这是为了避免软件在修改过程中对原有正常软件部分造成损坏。

回归测试的依据应根据测试的对象来确定，除了与相应级别的测试依据一致外，还需要依据被测软件的更动说明，根据软件的更动情况确定测试内容、测试策略和测试方法等。

严格地说，回归测试不是一个测试阶段，它可以用于单元测试、集成测试、配置项测试和系统测试等各个测试过程。回归测试与其他级别测试的区别，主要表现在以下 3 个方面。

（1）测试对象不同。回归测试更多地关注被变更的部分。

（2）测试的依据不同。除了依据相应级别测试所需的测试依据外，还需要依据软件更动说明，应根据更动说明进行更动影响域分析，以便确定软件回归测试范围。

（3）测试内容存在差异。回归测试的内容主要是软件变更部分和受影响的部分，而各级别的测试则是相应级别下较为全面的测试。

## 8.1.3 回归测试的重要性

软件测试后发生更动在所难免，在较为常用的迭代开发、极限编程和敏捷开发方法中，软件更动更会经常发生。更动可能带来程序代码的修改、新增或删除，而这些更动极有可能引入新的错误，从而使被测软件原有的正常功能出现新的错误。在重大工程中因软件更动造成的重大损失屡见不鲜，导致阿丽亚娜 5 型火箭发射失败的软件缺陷就是由于复用的代码没有经过充分的回归测试而造成的。因此，为了发现更动后软件存在的各种缺陷，纠正软件缺陷，有效控制和保证软件产品质量，就必须进行充分的软件回归测试。

如何在时间紧、更动量大的情况下提高回归测试对更动验证的针对性，保证回归测试的充分性及高效性是摆在测试者面前的一个难题。为此，本章对软件更动类型及特点、软件回归测试策略、软件更动影响域分析方法、回归测试用例设计策略及回归测试过程等内容进行了探讨与说明。

## 8.2　回归测试策略

软件回归测试的策略总体上分为两种：一是将更动前测试中包含的全部测试用例重新执行一遍，并增加一些针对新增功能的测试用例；二是根据软件变更的情况，对更动影响域进行分析，并在此基础上，设计回归测试用例。

第一种回归测试策略风险最小，这种方式将已有的测试用例与针对修改或扩充部分新设计的测试用例一并进行完整的测试。其优点在于对更动影响域定性分析的准确性要求较低，且完整的测试可以尽量多地发现受影响的部分；缺点是虽然遗漏错误的风险被降到最低，但是回归测试预算和进度的成本将大大增加，不利于回归测试活动的高效开展。当被测软件受更动影响的部分无法准确确定，并且测试时间较充分时可采用这种方式。

第二种回归测试策略是需要进行准确的软件更动影响域分析，根据分析结果针对变更所涉及内容设计相应的测试用例并进行测试。其优点在于测试代价较小，更动测试针对性强；缺点是对更动影响域定性分析的准确性要求较高，回归测试的有效性、充分性难以保障。

为了回归测试的高效开展，一般情况下，在高层次的测试中，例如集成测试、配置项测试、系统测试中，回归测试都采用第二种策略。该策略是将回归测试范围限定于软件的更动影响域之内，通过对软件更动前测试的用例集合进行拣择，选择其测试范围覆盖更动影响域的测试用例集，并在此基础上有针对性地增加一些新的测试用例构成回归测试用例集合。基于这样的用例集合实施回归测试，是一种较为简捷、充分且高效的回归测试策略。

## 8.3　软件更动影响域分析方法

在上述软件回归测试策略中，分析软件的更动情况，准确识别出更动所涉及的影响域范围，是保证软件回归测试充分开展的基础和重要环节，因此应根据软件变更情况开展有效的影响域分析，为回归测试设计奠定坚实的基础。更动影响域分析方法大体上可以分为两类：黑盒测试更动影响域分析方法和白盒测试更动影响域分析方法。

### 8.3.1　黑盒测试更动影响域分析

在黑盒测试中，软件更动类型可能包含 4 种类型。

（1）纠错性更动。纠正在软件开发过程中发现的错误，包括：

① 设计错误；

② 程序错误；

③ 数据错误；

④ 文档错误。

（2）适应性更动。为适应软件运行环境改变而进行的更动，包括：

① 影响系统的规则或规律的变化；

② 硬件配置的变化，如机型、终端、外部设备的改变等；

③ 数据格式或文件结构的改变；

④ 软件支持环境的改变，如操作系统、编译器或实用程序的变化等。

(3) 完善性更动。为扩充功能或改善性能而进行的更动，包括：

① 为扩充和增强功能而作的更动，如扩充解题范围和算法优化等；

② 为改善性能而作的更动，如提高运行速度、节省存储空间等；

③ 为便于维护而作的更动，如为了改进易读性而增加一些注释等。

(4) 预防性更动。在问题发生之前，为防止问题发生所进行的更动。这类更动的特点是没有“实际发生过的问题”的问题报告单。因此对于这类更动申请时可不需要软件问题报告单，但在软件更动申请及其附加报告中要更加详细地分析和说明更动的原因及预期结果。预防性更动主要包括：

① 在吸取其他软件的经验教训的基础上或对其他发生过的问题“举一反三”后，为预防问题的发生对软件进行更动，如增加软件防“跑飞”措施等；

② 为改进软件的可维护性或可靠性，或者为了给未来的改进奠定更好的基础而更动软件，如采用逆向工程与重构工程等先进技术更动或重构已有的系统，产生一个新版本。

软件更动完毕后，软件更动方应提交软件更动说明对更动项进行描述。其中虽然会给出更动涉及的软件需求描述，但一般都比较粗略，而且依据该信息，测试人员很难准确定位到前次测试中的具体软件需求依据，也必将影响到后续对需求依据所涉及测试用例的拣择范围。因此，在软件更动说明所描述信息的基础上，测试人员应进行进一步的回归测试更动影响域分析，准确获取每一处更动涉及的前次测试的软件需求项。

黑盒测试的软件更动影响域分析可采用下述测试信息多层次结构模型及针对各类更动的影响域分析方法。

**1. 测试信息多层次结构模型**

由于测试信息多层次结构模型依赖于项目初次测试的信息层次，因此先将项目初次测试的流程以及信息层次做一介绍。

1) 初次测试流程

项目进行黑盒测试一般依据的是被测软件的需求规格说明文档，其主要流程如下。

首次测试中测试方首先根据被测软件需求文档中针对各功能、性能、接口及其他方面的文本描述进行项目首次测试的需求梳理，梳理出的需求项将作为首次测试的测试依据。接下来测试人员依据被测软件的需求，确定被测对象及其测试类型。之后测试人员应针对每个测试类型确定其包含哪些测试点(如功能测试的测试点就是该功能所包含的所有功能细项)。测试人员应针对每个测试点分别设计一个测试项，明确每个测试项的测试要求、测试策略及方法。此外，还应在测试项与软件需求项之间建立双向追踪关系，以保证全面的测试覆盖及避免冗余测试的发生。上述工作完成后，测试人员将为每个测试项设计测试用例。在之后的测试用例执行过程中，测试人员记录测试结果并提交测试过程中所发现的问题。

当首次测试结束后，被测方将针对测试中发现的问题进行分析，查找其原因并进行程序的更动。更动后被测方应提交软件更动说明，说明所做更动的原因、更动前后的代码对比、更动涉及的软件需求。基于软件更动说明，测试方将对被测软件进行相应的回归测试活动。

2）初次测试的信息分类

（1）测试需求。测试需求是软件测试的测试依据。黑盒测试的测试需求一般为被测软件需求规格说明文档中所明确的各个要求。测试人员应对被测软件的需求规格说明文档进行梳理，梳理出各个需要测试的要求，并分别以软件需求项的形式表示。

（2）被测对象。在软件测试过程中进行独立测试并进行质量评估的一个软件实体，如软件单元、软件部件、软件配置项等都是软件测试的被测对象。一个测试项目至少包含一个被测对象。

（3）测试类型。测试类型是对被测对象测试内容的逻辑划分或组合，如将测试内容划分为功能测试、性能测试、接口测试、安全性测试、可靠性测试等，软件测试实践中存在着多种测试类型的划分或组合方法。一般情况下，将被测对象不同性质的一个或一组物理属性归属到某个测试类型。一个被测对象的测试内容至少被划分或组合为一个测试类型。

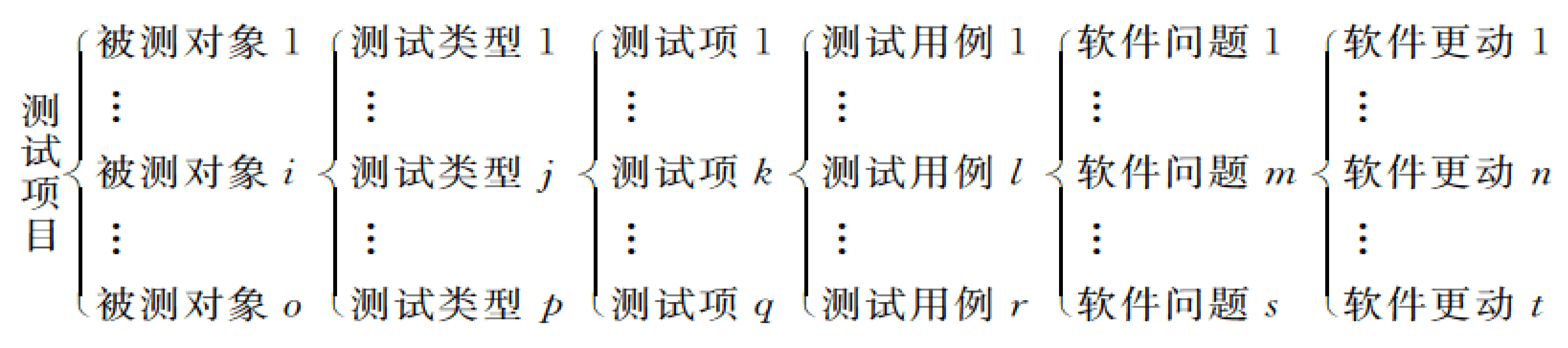
（4）测试项（测试条目）。测试项是被测对象某个测试类型所包含的各个测试点的描述，如被测对象功能测试类型下的某个测试项就是针对某个子功能的测试要求描述。测试项也可被称为测试条目。一个被测对象的某个测试类型中至少包含一个测试项（测试条目）。测试项实际上是针对梳理出的各软件需求项的测试实施点。各测试项应建立与上述（1）中梳理出的相应软件需求项之间的关联关系，每个测试项至少应与一条软件需求项建立联系。

（5）测试用例。测试用例是支持测试项（测试条目）的具体测试方法的具体描述，主要包括测试用例名称、标识、测试初始条件、测试步骤集（每一步骤的测试输入、期望结果、判别标准）、测试用例通过准则等要素。一个测试项（测试条目）下至少应有一个软件测试用例。

（6）软件问题。软件问题是执行测试用例过程中发现的实测结果与期望结果之间存在不一致情况的描述。一个测试用例可提交一个或多个软件问题。

（7）软件更动。此处的软件更动是为了解决前次测试所发现软件问题所进行的更动，每个软件问题可能发生一个或多个软件更动。

被测对象、测试类型、测试项、测试用例、软件问题、软件更动的层次关系如下所示，测试项目的初次测试可按照这样的层次结构组织测试信息。梳理出的软件需求项因为与测试类型之间没有层次关系，因此未列在下述层次中，但其与测试项之间却存在关联关系。




3）信息多层次结构模型

我们将测试信息进行分类，利用黑盒测试首次测试中的软件需求项、测试项、测试用例、软件问题、软件更动之间的关联关系，建立一个多层次结构模型对上述测试信息进行组织及管理。软件需求项位于该层次结构模型的最高层，所有测试信息以它为基础呈树状展开；所有软件更动位于该层次结构模型的最底层，是进行回归测试更动影响域分析的源头。测试信息多层次结构模型如图 8-1 所示。

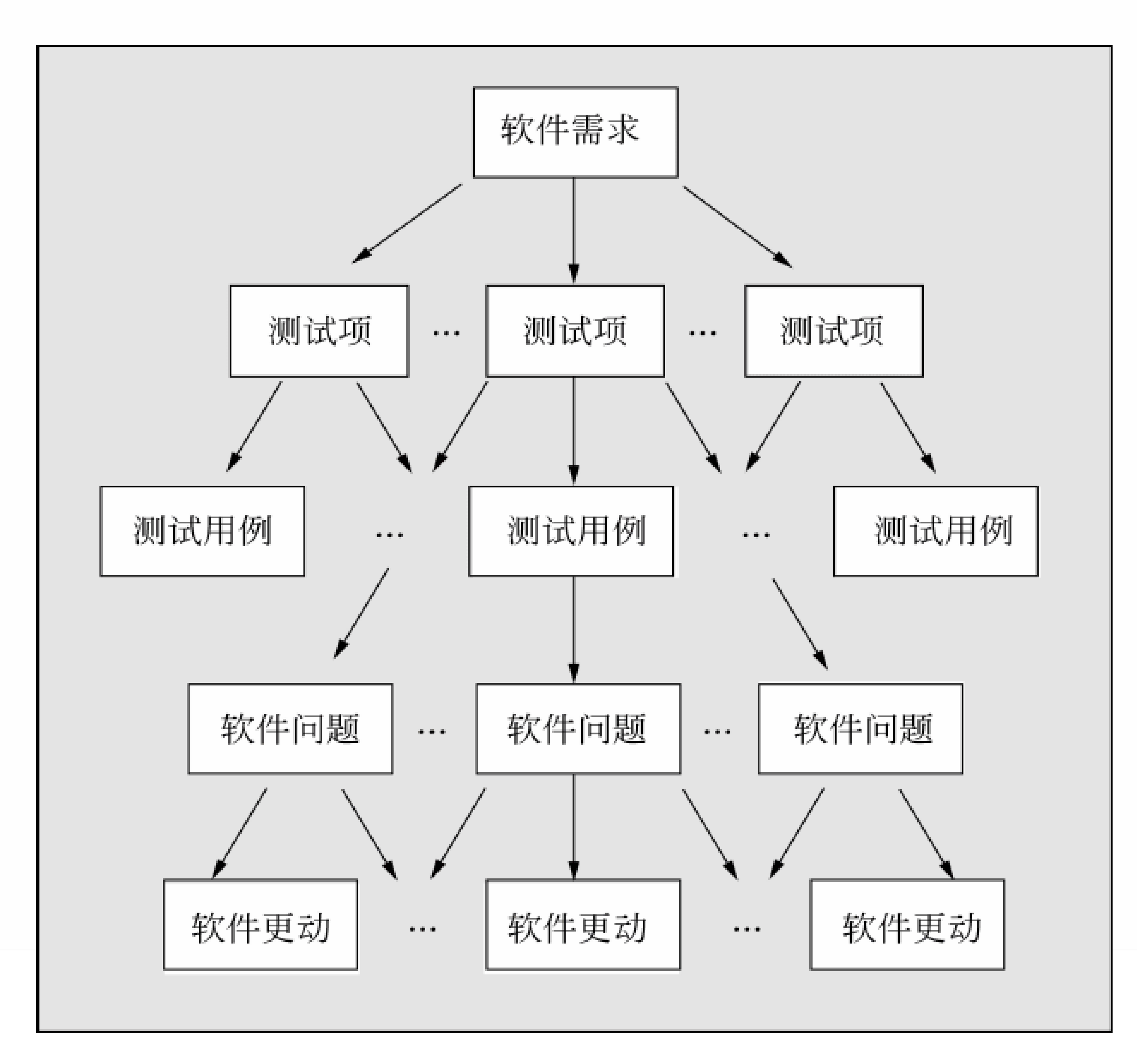



**图 8-1 测试信息多层次结构模型图**

基于上述测试信息的多层次结构模型,我们分别为各类更动提出了相应的更动影响域分析方法。

**2. 各类更动的影响域分析方法**

1) 纠错性更动

此类更动是为了解决前次测试发现问题所进行的更动。针对此类更动,可利用上述首次测试信息多层次结构模型中各层次数据之间的关联关系,按照由底向上、逐层追溯的原则,获取每个软件更动所涉及的软件需求项。

按照上述方法,首先定位处于测试信息多层次结构最底层的某个软件更动,然后由底向上,依次追溯到该更动所针对的软件问题、提交问题的测试用例、用例所属测试项、与测试项所关联的软件需求项集合,该集合就是纠错性更动所涉及的更动影响域范围。

2) 适应性更动

此类更动是根据软件运行情况的变化而进行的更动,因此大多是对软件需求的变更或增减。针对此类更动应依据软件更动方提交的更动说明,进行以下分析:

(1) 对其变更的需求进行分析,对已有的测试需求项进行修改;

(2) 对其中新增需求进行梳理,分解为多个软件需求项;

(3) 而针对软件功能删减的情况,测试人员应在前次测试所梳理的测试需求依据中将删减部分涉及的软件需求项进行相应删除。

3) 完善性更动

此类更动是由于软件实现被优化所引起的,与软件问题没有关联。其更动涉及的软件需求内容依赖于提交的更动信息。因此,应先在软件更动说明中找到关于此类更动所涉及需求内容的相应描述,确定此类更动所涉及的前次测试的软件需求项集合。

4）预防性更动

预防性更动一般是在原有软件需求的基础上增加新的需求。针对此类更动应依据提交的更动说明，对其中新增需求进行梳理，分解为多个软件需求项。

经过针对上述 4 种更动情况的影响域分析，在黑盒测试进行回归测试时，其更动所涉及的影响域范围被最终确定下来。

## 8.3.2　白盒测试更动影响域分析

白盒测试认为软件更动实际上是修改了程序的定义和使用关系，因此白盒测试进行回归测试时关注的是更动后被测软件程序中各定义和使用关系的正确性。如果将软件更动前所有用以验证各定义和使用关系的测试用例全部重新执行，测试预算和进度的成本必将大大增加，显然不利于白盒回归测试活动的高效开展。为了解决上述问题，就要求测试人员必须准确获取软件更动影响到的程序中的定义和使用关系，并针对该影响域范围展开相应的回归测试活动。

**1. 软件更动类型及影响域**

受更动影响的定义和使用关系可分为两大类：一类是因为定义或者使用的增减而带来的直接影响，即产生了新的定义和使用关系；另一类是因为变量值的改变或者条件语句中谓语的改变而对定义和使用关系产生的间接影响。因此，白盒测试中软件更动影响域分析就是要分析出程序更动后受到直接或间接影响的定义和使用关系。

为此，首先将回归测试可能出现的软件更动类型进行归纳，并针对每一种更动对定义和使用关系的影响进行分析。

更动对程序所带来的影响可归纳为以下 3 种类型。

（1）产生了新的定义和使用关系。引发这种情况一般是由于定义或者在语句中增加了对别的变量的引用。如下面的代码段：

```
1 if (A MOD 3) then
2  X:=5
3  end if
4  Y:=2
```

如果上述代码段的语句 4 被“Y：＝2＋X”代替，就引进了新的变量 $X$，从而产生了新的定义和使用关系。

（2）因为变量值的改变而使得一些定义和使用关系受到间接影响。这种情况下不产生新的定义和使用关系，但那些依赖于更动变量值的定义和使用关系会受到间接影响。如下面的代码段：

```
1 X:=10
2  if (A >1) then
3  Y:=X +5
4  End if
```

如果上述代码段的语句 1 被“X：＝20”代替，虽然没有产生新的定义和使用关系，但由于变

量 $X$ 的值发生变化，使所有依赖于 $X$ 的值的定义和使用关系也发生变化。

(3) 因为条件语句中谓词的值或操作符的改变而使得一些定义和使用关系受到间接影响。这种情况下不产生新的定义和使用关系，但是所有控制依赖于谓词的值或操作符发生改变语句的定义和使用关系都会受到影响。如下面的代码段：

```
1 X:=10
2  if (A >1) then
3  Y:=X +5
4  End if
```

如果上述代码段的语句 2 被“if(A<1)then”代替，尽管没有产生新的定义和使用关系，但所有控制依赖于语句 2 的定义和使用关系都受到影响。

**2. 更动影响域分析方法**

程序切片技术是一种用于分解程序的分析技术，其目的是获取影响程序某关注变量的所有语句和控制谓词组成的集合。任何一个程序可以和一组程序切片的并集等价，测试一个程序就可以转化成测试该程序的一组切片及其补集，从而保证了软件测试具有充分的覆盖。由于回归测试中获取更动影响域是要得到由于更动产生的所有受到影响的定义和使用关系集合，与程序切片技术的目的一致，因此可以利用程序切片的思想来获取软件更动的影响域。

目前可通过程序切片的方法获取受更动影响的定义和使用关系。该方法的主体思想是：首先从程序更动处开始沿着后向进行遍历，找到与被更动的语句有关的定义点；再从程序更动处开始沿着前向进行遍历，寻找定义点因为程序更动而发生变化后所影响到的所有使用点及其定义点。

该方法是基于控制流图的，应先画出软件更动前程序的控制流图。在控制流图中每个节点代表一条语句，边代表语句之间的控制流，变量的定义和使用信息附加在每个节点旁边。分析完成后得到的结果是受更动影响的所有定义和使用关系的集合，其中每个定义和使用关系用三元组 $\langle s,u,v\rangle$ 表示，表示在语句 $s$ 中定义的变量 $v$ 在语句 $u$ 中被使用。

上述方法的主体思想可以流程图的形式进行展现，如图 8-2 所示。

该方法的实施步骤如下：在被测程序的控制流图中，某个更动节点为 $n_i$，其更动变量为 $X$。首先沿着控制流图的逆向进行遍历，查看遍历经过的节点中是否包含更动变量 $X$ 的定义节点。如果节点 $n_j$ 包含对该更动变量 $X$ 的定义，则将定义节点 $n_j$ 到更动节点 $n_i$ 的定义和使用关系记录下来，记为 $\langle n_j, n_i, X\rangle$；然后将节点 $n_i$ 及该语句节点中由于变量 $X$ 的更动而受到影响的变量 $Y$ 放入一个列表中，然后从节点 $n_i$ 开始沿着控制流图的正向进行遍历，查看控制流图中各节点旁边的引用变量是否包含变量 $Y$，如果节点 $n_k$ 的引用变量包含该变量，则将节点 $n_k$ 及受引用 $Y$ 而影响到的变量 $Z$ 加入列表，并将受影响的定义和使用关系记录下来，记为 $\langle n_i, n_k, Y\rangle$。当针对节点 $n_i$ 中变量 $Y$ 引用情况的遍历结束后，将节点 $n_i$ 及变量 $Y$ 从列表中去除，按照列表中元素的先后顺序，依次查找各引用节点中相应变量的节点，将受影响节点及变量依次加入列表，并将受影响的定义和使用关系记录下来。重复上述过程，直至列表中所有元素的影响范围信息均遍历完成，最后得到的结果为一个受更动影响的定义和使用关系集合。

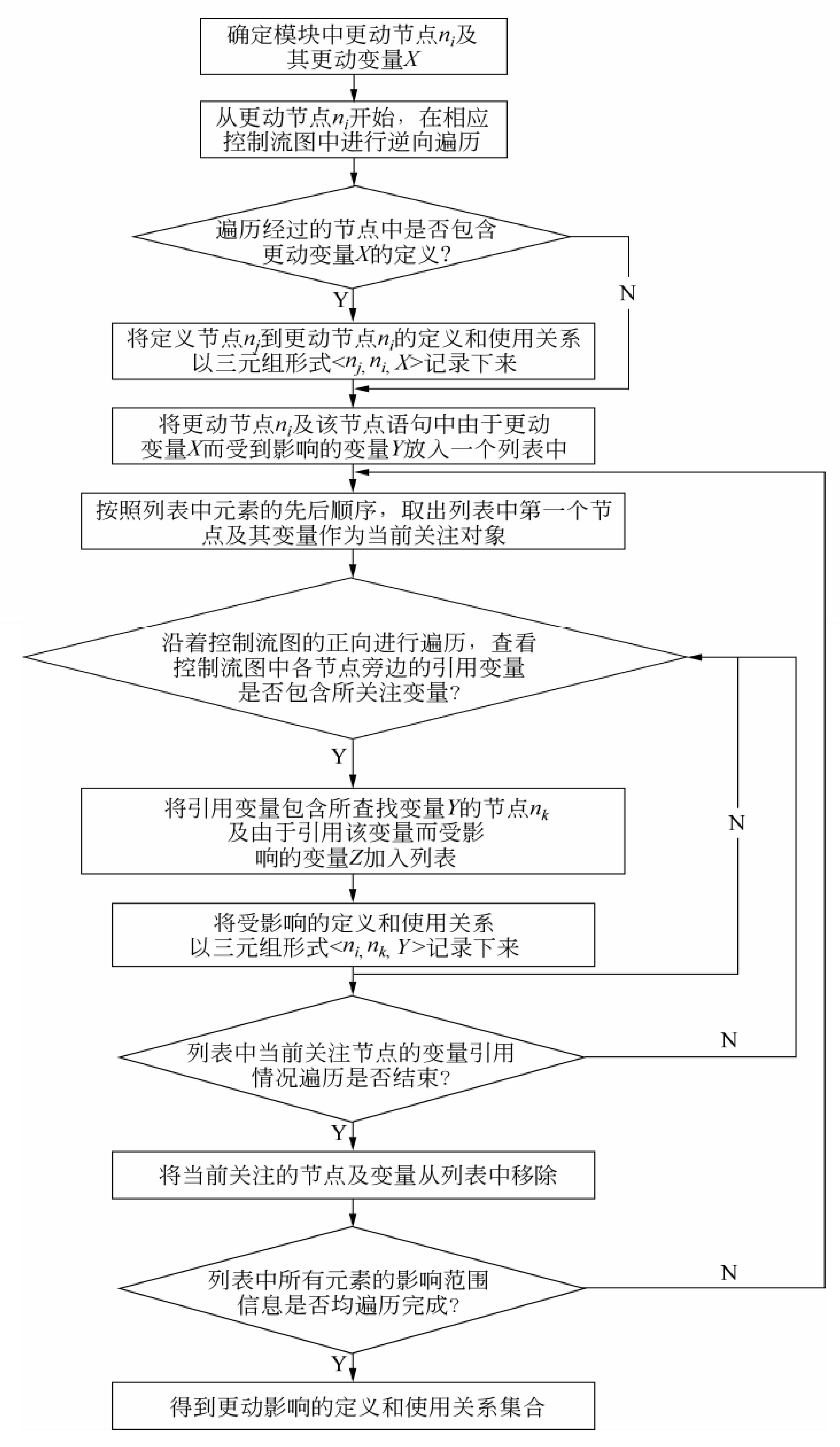
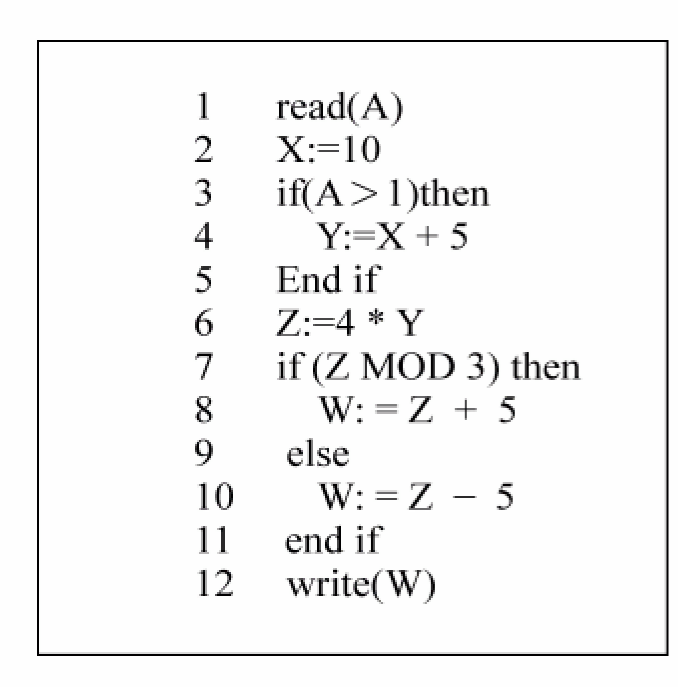



**图 8-2　更动影响域分析方法流程图**

### 3. 实例展示

下面以一段程序代码为例，在该段程序代码中设置了上述 3 种更动类型，按照上述切片方法逐一对其影响域的获取方法进行具体说明。图 8-3 是一段程序源代码，图 8-4 是此段代码的控制流图。每个节点代表一条语句，每个语句中用到的变量标注在节点旁边。



```
1   read(A)
2   X:=10
3   if(A > 1)then
4      Y:=X + 5
5   End if
6   Z:=4 * Y
7   if (Z MOD 3) then
8      W: = Z + 5
9    else
10     W: = Z - 5
11   end if
12   write(W)
```

图 8-3 程序源代码示例

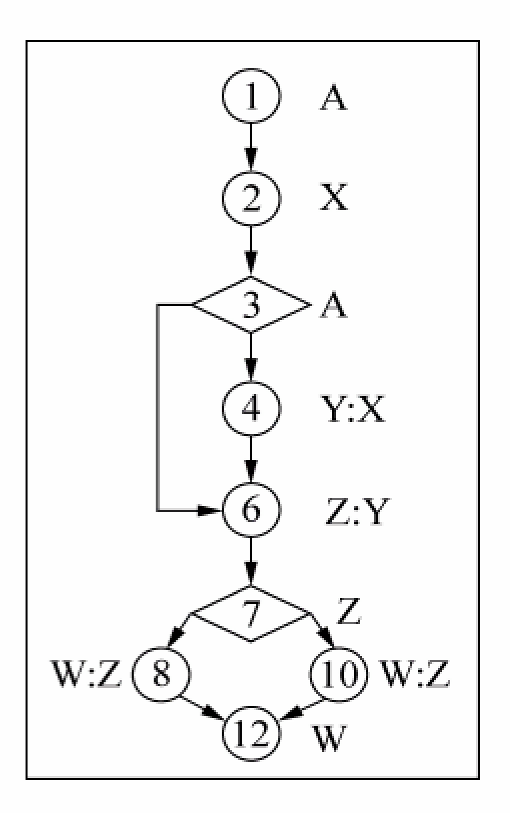



图 8-4 程序源代码示例的控制流

1）产生了新的定义和使用关系

上述程序中语句 6 被“Z：＝4 ∗（Y＋X）”代替，引进了新的变量 X。

首先，沿着控制流图中控制流的逆向查找 *X* 的定义点，该定义点在语句 2 所对应的节点处被找到。由于语句 6 中引进了新的变量 *X*，因此从语句 2 到语句 6 之间就增加了新的定义和使用关系〈2，6，*X*〉；然后从语句 6 所在节点开始，沿着控制流的正向查找此更动对后续的影响。由于语句 6 更动了变量 *Z* 的值，而语句 8、10 用到变量 *Z*，因此从语句 6 到语句 8、10 的关于变量 *Z* 的定义和使用关系〈6，8，*Z*〉、〈6，10，*Z*〉受到间接影响，并且语句 8、10 中的变量 *W* 也受到影响，而语句 12 用到了变量 *W*。因此从语句 8、10 到语句 12 的关于变量 *W* 的定义和使用关系〈8，12，*W*〉、〈10，12，*W*〉受到间接影响。

基于控制流图，利用切片方法分析得到语句 6 发生上述更动后，影响到的定义和使用关系包括：〈2，6，*X*〉、〈6，8，*Z*〉、〈6，10，*Z*〉、〈8，12，*W*〉、〈10，12，*W*〉。

2）因为变量值的改变而使得一些定义和使用关系受到间接影响

上述程序中语句 2 被“X：＝20”替换，所有依赖于语句 2 中 *X* 值的定义和使用关系都受到了影响，但没有产生新的定义和使用关系。在控制流图中的语句 2 对应节点开始，沿着控制流图的正方向按照切片方法进行查找，最终获取到受影响的定义和使用关系包括：〈2，4，*X*〉、〈4，6，*Y*〉、〈6，8，*Z*〉、〈6，10，*Z*〉、〈8，12，*W*〉、〈10，12，*W*〉。

3）因为条件语句中谓词的值或操作符的改变而使得一些定义和使用关系受到间接影响

上述程序中语句 3 被“if（A ＜ 1）”代替，所有依赖于语句 3 的定义和使用关系都会受到影响，但没有产生新的定义和使用关系。这些受影响的定义和使用关系有〈4，6，*Y*〉（因为语句 4 控制依赖于语句 3），还有后续受影响的定义和使用关系，包括：〈6，8，*Z*〉、〈6，10，*Z*〉、〈8，12，*W*〉、〈10，12，*W*〉。

上述 3 种情况分别使用程序切片方法获取到各自更动受直接影响或间接影响的定义和使用关系，即获取到各种更动的相应影响域。

通过上述程序切片的方法获取白盒回归测试中软件更动的影响域，缩小了白盒回归测试中软件测试的范围，显著提高了回归测试的效率；并且由于通过程序切片技术获取白盒更动影响域的方法考虑到程序中存在的各种依赖关系，使得获取的更动影响域更加全面、准

确，能够有效保证被测软件白盒回归测试的充分性。

# 8.4　回归测试用例设计

## 8.4.1　回归测试用例设计原则

回归测试用例设计包括两部分，一是更动前测试所包含测试用例的选取，二是新增测试用例的设计。新增测试用例的设计与其他测试没有太大区别，但如何在更动前测试所包含测试用例的大集合中，合理选取涉及软件更动部分及受更动影响部分的测试用例却存在一定的困难。

常用的测试用例选取策略有下述 3 种。

(1) 选取软件更动范围内的测试用例。该原则是依据修改的内容来选择测试用例，这些测试用例仅用来验证所修改软件部分的实现是正确的。这种方法的效率最高，然而风险也最大。因为它没有考虑某一个修改是否影响了软件的其他部分。这种方法适合用于测试时间短，软件结构耦合性很小的情况下进行回归测试时用例的选取。

(2) 选取受影响功能的测试用例。该原则需要根据影响域分析的结果，选取所有受影响部分所涉及的测试用例。对于不同级别的测试，选取受影响功能的测试用例方法也不一样，多数时候还需要依靠测试人员的经验。例如，对于单元测试来说，回归测试需要考虑到软件的更动可能对公共接口的影响，特别是全局变量、输入输出接口、配置文件等。对于配置项和系统测试来说，回归测试需要考虑到修改部分对性能和接口的影响等。

(3) 规定回归测试用例指标。该原则一般用于在相关受影响部分难以界定的时候使用。一般由测试人员与开发人员共同确定测试用例选取的指标。例如：规定修改范围内的测试用例全部选取，其他测试用例选取 60%。

## 8.4.2　已有测试用例的选取

综合上述原则，可以制定回归测试设计时对已有测试用例的选取策略。该策略是基于测试需求的追踪关系而建立的。软件测试过程中的追踪关系可表示为如图 8-5 所示的形式。

**1. 测试需求追踪关系定义**

测试需求的追踪关系包括下列一组追踪关系。

1) 测试需求间的关系

软件测试需求集定义为 $R=\{r_1,r_2,r_3,\cdots,r_i,\cdots\}$，是被测软件系统需要覆盖的测试需求集合。测试需求集是软件的组成模块，如从结构角度一个测试需求可以是一条语句或一个函数(过程或方法)。从功能角度一个测试需求可以是一个功能项或一个被测特征。

在进行测试需求分析时，依据软件需求规格说明确定测试需求。测试需求包括软件的功能、性能、接口、人机界面、安全性、可靠性和可测试性要求等。

测试需求之间存在着关联关系。在进行测试需求分析时，可以根据软件需求规格说明

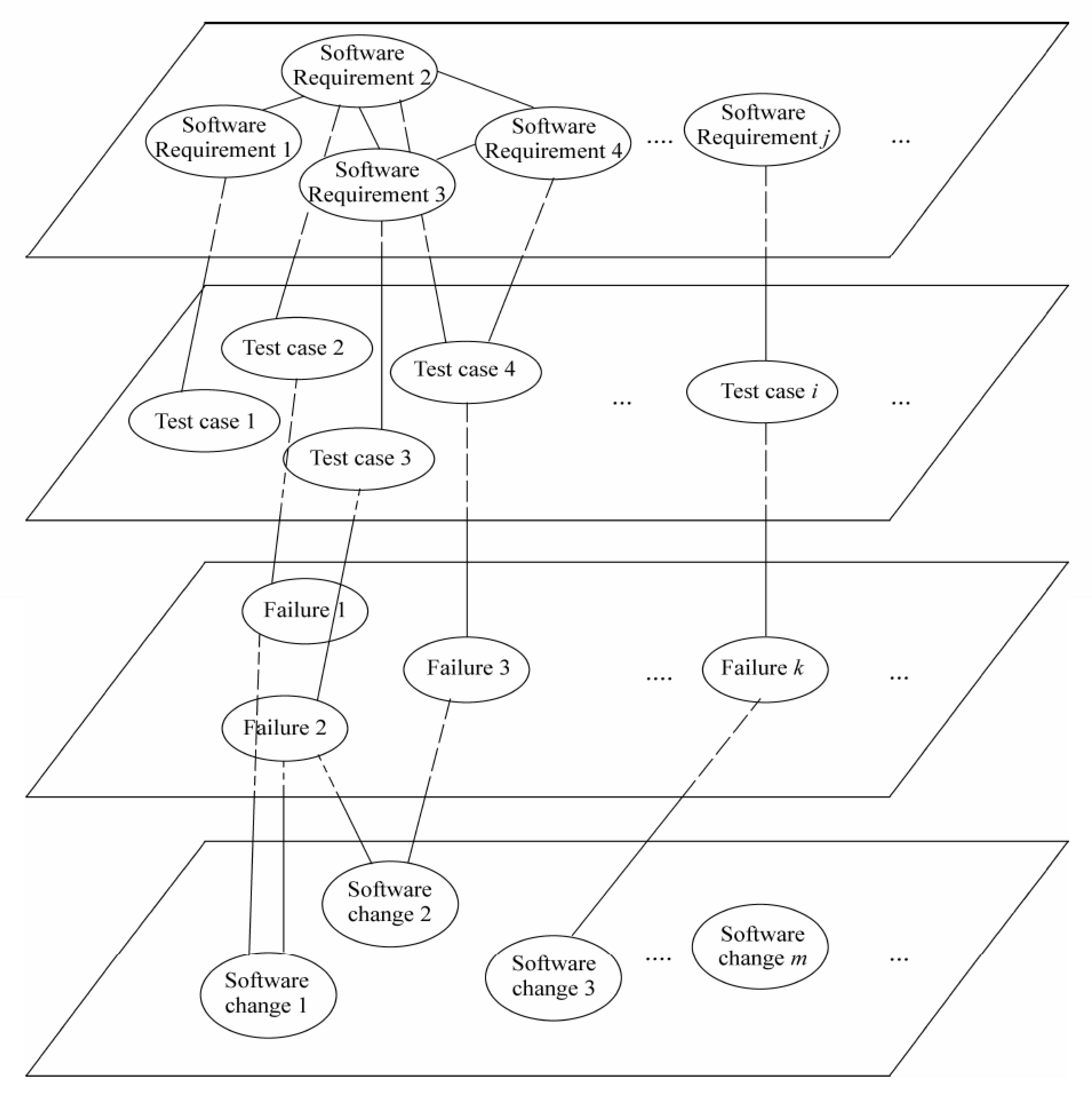



**图 8-5　软件测试中的追踪关系示意图**

得到软件需求间的关联关系。在采用结构化方法描述的软件需求规格说明中，可根据软件数据流图和控制流图确定软件需求之间的关系。在采用面向对象方法描述的软件需求规格说明中，可根据用例图确定软件需求间的关联关系。考虑实际应用的便捷性，可以将软件测试需求之间的关联关系表达为无向联通图。

2）测试需求与测试项之间的追踪关系

测试项的集合定义为 $I=\{item_1, item_2, item_3, \cdots, item_j, \cdots\}$，是对被测软件确定的测试项。测试项是为了有效实施对某一软件需求的测试而设计的测试要求。测试项的描述包括名称、标识、测试项说明、测试方法、测试充分性要求、测试项终止要求、测试项优先级以及测试项对测试需求的追踪关系。

分析测试需求与测试项的关系可以得到：一个测试需求由一个或多个测试项覆盖，一个测试项关联一个或多个测试需求。

3）测试项与测试用例的追踪关系

软件测试用例集定义为 $T=\{t_1,t_2,t_3,\cdots,t_k,\cdots\}$，是针对被测软件设计的一组测试用例集合。分析测试用例集与测试项集之间的关系，可以得出以下结论：一个测试用例可能覆盖一个或多个测试项，同时一个测试项可能需要一个或多个测试用例来完成测试要求。

4）测试用例与软件问题的追踪关系

软件问题集定义为 $P=\{p_1,p_2,p_3,\cdots,p_l,\cdots\}$，是执行软件测试用例后标识出的软件失效，这些失效是软件缺陷的外在体现。因此，一个软件问题一定与某一个测试用例相关联。也就是说，测试发现的软件问题，一定是执行某个用例而产生的。这样，从一个问题一定能够追踪到发现它的测试用例。

5）软件问题与软件更动之间的追踪关系

软件更动集定义为 $C=\{c_1,c_2,c_3,\cdots,c_m,\cdots\}$，除了是针对软件问题进行的修改以外，还包括适应性、完善性和预防性更动。对于软件变更不仅需要对提交问题的用例进行重新执行，还需要根据影响域分析的结果选择已有的用例或新增用例进行测试。对于新增测试需求一般情况下需要增加新的用例进行测试。

分析软件问题与软件更动之间的关系，可以得到以下结论：一个软件问题对应一个或多个软件更动，而一个软件更动也可以对应一个或多个软件问题。而考虑新增的软件需求或由于适应性修改需求引起的变更，同时考虑方法实现的便捷性，可以用软件更动对应零个问题来进行描述。

**2. 选取方法相关定义**

针对上述追踪关系分析，可以给出如下相关定义：

**定义 1** 原测试版本的测试需求集为 $R$，测试项集为 $I$，测试用例集为 $T$。回归测试的测试需求集为 $R'$，测试项集为 $I'$，测试用例集为 $T'$，软件问题集为 $P$，软件更动集为 $C$。

**定义 2** 软件测试需求 $r_i$ 的属性为名称、标识、说明。

**定义 3** 软件测试项 $i_j$ 的属性为名称、标识、测试项说明、测试方法、测试充分性要求、测试项终止要求、测试项优先级以及测试项对测试需求的追踪关系。其中，设测试项的优先级为：高、中、低。

**定义 4** 软件测试用例 $t_k$ 的属性为：测试用例名称、标识、用例综述、对测试项的追踪关系、测试用例优先级、初始化要求、前提和约束、测试步骤、用例的期望结果、测试结果评估准则等。其中，设测试用例的优先级为：高、中、低。

**定义 5** 软件问题 $p_l$ 的属性为：问题标识、问题类别、问题级别、关联的测试用例、问题描述和改进意见等。

**定义 6** 软件更动 $c_m$ 的属性为：更动标识、更动说明、更动类型以及更动影响域分析说明。其中，更动类型为“0”时，该软件更动为纠错性更动；当更动类型为“1”时，软件更动为完善性更动；当更动类型为“2”时，软件更动为适应性更动；当更动类型为“3”时，软件更动为预防性更动。

**3. 已有测试用例选取算法**

根据影响域分析结果进行的软件回归测试用例选取算法如下。

（1）当软件更动的更动类型为“0”时，应执行以下步骤进行回归测试用例的选取：

① 遍历软件更动集合中的所有更动，依据变更对问题的追踪关系，获取变更所对应的问题，并以此为线索，查找问题所对应的用例，将其放入回归测试用例集 $T'$ 中；

② 遍历测试用例集 $T'$ 中所有测试用例，依据用例对测试项的追踪关系，获取所有相关的测试项，放入 $I'$ 中；

③ 遍历 $I'$ 中所有测试项，依据测试项对测试需求的追踪关系，获取所有相关的测试需求，放入 $R''$ 中；

④ 先将 $R''$ 中所有测试需求放入 $R'$ 中，之后遍历 $R''$ 中所有测试需求，将与 $R''$ 中每个测试需求相关的其他测试需求也放入到回归测试需求集 $R'$ 中；

⑤ 遍历 $R'$ 中所有测试需求，将与 $R'$ 中每个测试需求相关的高优先级测试项，且在 $I'$ 中不存在的测试项添加入 $I'$ 中；

⑥ 遍历 $I'$ 中所有测试项，将与 $I'$ 中每个测试项相关的高优先级测试用例，且在 $T'$ 中不存在的测试用例添加入 $T'$ 中。

上述测试用例选取算法可表示为如图 8-6 所示。

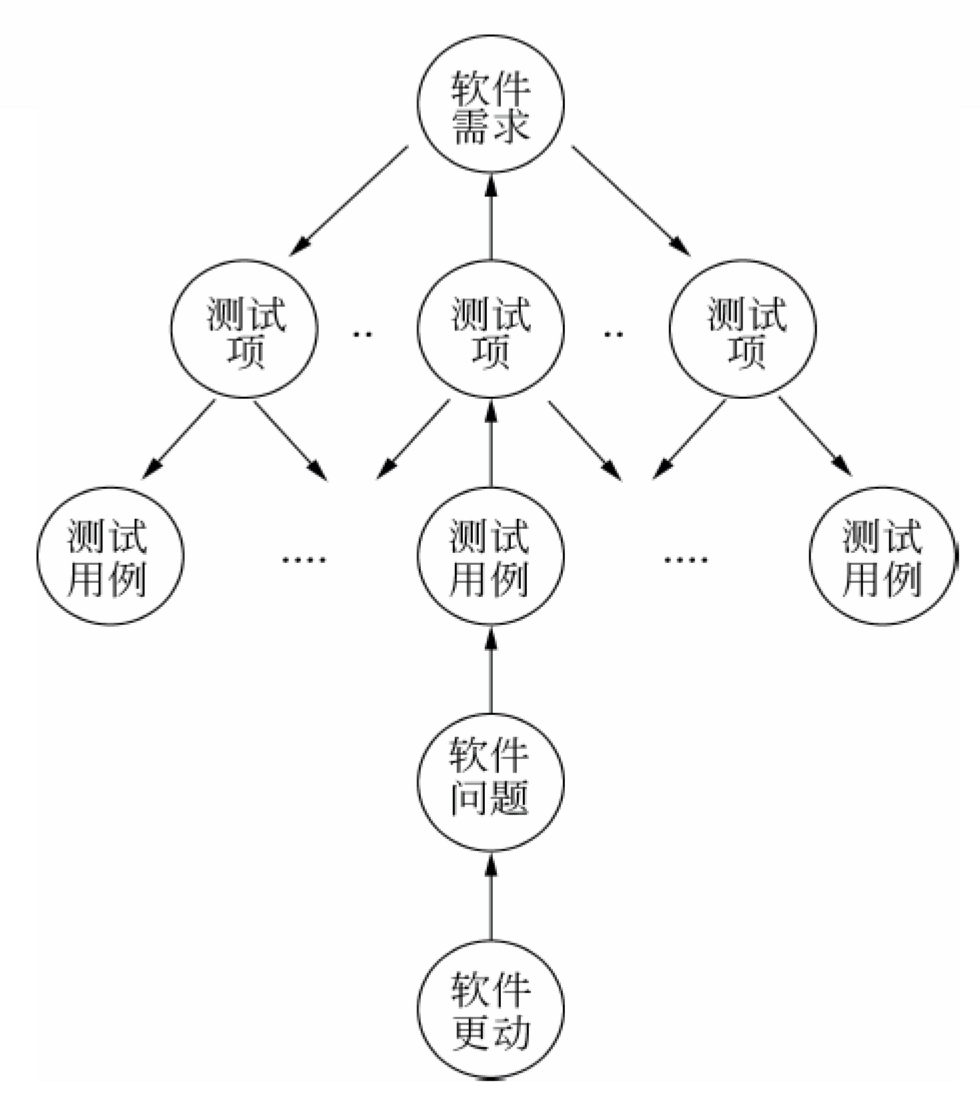



**图 8-6 纠错性更动的回归测试用例选取算法示意图**

（2）当软件更动的更动类型不为“0”时，将软件更动的要求放入测试需求 $R'$ 中。

上述(1)中各步已完成了回归测试中对已有测试用例的筛选。在(2)中需要软件测试人员为每个新增测试需求设计新的测试项，并为每个新增测试项设计新的测试用例。这些测试用例应覆盖新增需求对软件各特性的要求，以保证新增测试设计的充分性。

## 8.5 回归测试过程

软件回归测试过程往往与某一级别的测试执行阶段相互融合。例如在进行配置项测试首轮测试后，发生软件更动后即可进行回归测试。因此，回归测试过程较为简捷，但对测试

过程的主要要求是一致的。软件回归测试的过程如图 8-7 所示。

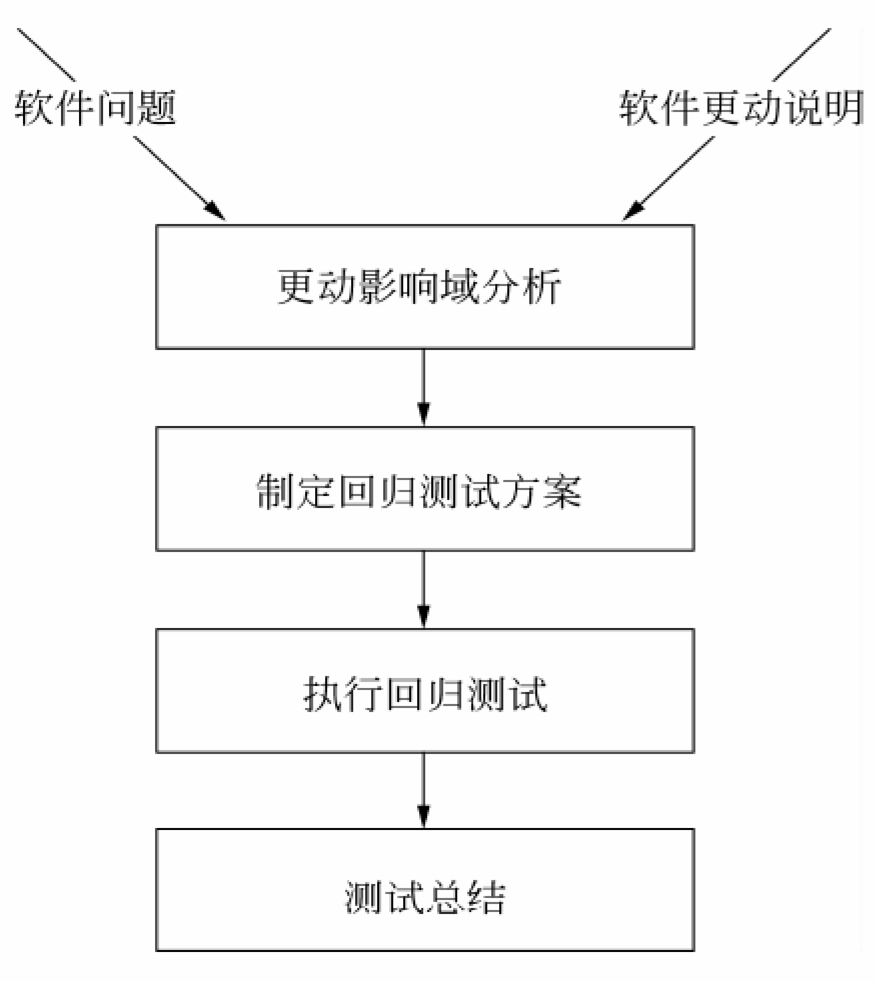



**图 8-7　软件回归测试过程**

#### 1. 更动影响域分析

该阶段依据软件的问题报告和软件更动说明，分析每项更动的影响域。在黑盒测试中获取更动影响域就是确定更动涉及的前次测试所依据的软件需求，在白盒测试中获取更动影响域就是确定被测软件更动后受到直接或间接影响的定义和使用关系，所获取的更动影响域将作为软件回归测试的测试需求依据。

#### 2. 制定回归测试方案

制定回归测试方案是根据更动影响域分析的结果确定回归测试需求、制定回归测试计划、设计回归测试用例的过程。其中测试计划、测试用例的内容要求与相应级别的要求一致。一般情况下，回归测试方案应进行评审，以保证回归测试方案的充分性。

#### 3. 执行回归测试

该阶段执行回归测试用例集合。测试过程中测试人员应客观记录每个回归用例的测试结果，提交回归测试过程中所发现的问题。

如果回归测试中仍然发现问题，并且被测方针对问题又进行了程序更动，那么更动后的软件应进行下一轮的回归测试。因此，回归测试有可能存在多轮次的情况，直至测试中已发现问题所引起的软件更动都通过测试，且不再发现新的问题。

#### 4. 测试总结

完成回归测试后，需要对被测软件经历测试的情况进行全面总结。当测试与回归测试的进度安排较为紧凑时，可将测试与回归测试的总结一并进行；而当回归测试与之前测试的时间间隔较长，需要出具相应的测试结论时，可以单独针对回归测试的情况进行总结。回归测试总结的内容要求与一般的软件测试报告一致。

# 第 9 章 CHAPTER 9

# 面向对象软件测试

当前，面向对象的软件开发方法已被广泛应用，人们对软件质量也提出了更高要求。面向对象软件的测试方法作为验证面向对象软件质量的主要手段，也相应得到了人们的广泛重视。面向对象的封装性、继承性、多态性等特性，提高了软件的可重用性，便于软件团队协作设计开发，而且易于维护和修改。但同时这些新特点也带来了新的风险，并给软件测试提出了新的挑战，使得传统的测试方法和技术已不能完全胜任面向对象的软件测试。

## 9.1 面向对象软件简介

面向对象软件开发是目前主流的软件开发技术，正代替传统的面向过程开发方法，逐渐成为主流的软件开发方法。面向对象技术产生更好的系统结构，更规范的编程风格，极大优化了数据使用的安全性，提高了程序代码的可重用性。

面向对象的程序设计是以下面的 7 个基本概念为核心的：对象、消息、接口、类、继承、封装和多态。

**1. 对象**

对象是一个可操作的实体，既包含了特定的数据，也包含了操作这些数据的代码。在面向对象的程序设计中，对象是一个基本的可计算实体，对象被创建、修改、访问或由于协作的结果而被删除。在一个良好的面向对象的设计理念中，程序中的对象是一些问题及其解决方法的特定实体的描述。在程序中，对象之间存在着一定的联系，这种联系反映了问题域中实体的相对关系。

对象是软件开发期间测试的直接目标。在程序运行时，对象的行为是否符合它的需求规格说明，该对象与相关的对象是否协同工作，这两方面是面向对象软件测试所关注的焦点。

对象可以用它的生命周期来描述。当一个对象被创建时它的生命周期就开始了，这个过程贯穿于对象的一系列状态，当一个对象被删除时它的生命周期也就结束了。

从测试视角的角度，关于对象可以得出如下观点。

（1）对象的封装。这使得已定义的对象容易识别，在系统中容易传递，也容易操纵。

（2）对象隐藏了信息。这一点使得对象信息的改变有时很难观察到，因此也使得测试结果分析的难度加大。

（3）对象的状态。在对象的生命周期中，对象都有一个状态。对象的状态是多变的，因

此也可能是不正常行为的根源。

(4) 对象的生命周期。对象都具有生命周期。在对象生命周期的不同阶段，为了确定对象的状态是否符合它的生命周期，对象可能会被从各个方面进行检测。过早地创建一个对象或删除一个对象，都是造成错误的常见原因。

**2. 消息**

消息是对象的操作将要执行的一种请求。除了需要一个操作的名字，消息还可包含一些值（实参），它们常常在操作被执行时使用。消息的接收者也可以将某个值返回给消息的发送者。

面向对象的程序是通过一系列对象协作工作来解决问题的，这一协作是通过对象之间互相传送消息来完成的。一般地，把发送消息的对象称为发送者，把接收消息的对象称为接收者。有一些消息会以不同的形式返回结果，例如接收者以返回值或者异常的形式把结果传送给发送者。

面向对象程序执行的典型过程，首先是实例化对象，然后将一条消息传送给其中的某个对象，消息的接收者把它自己产生的消息发送给其他对象（也可以发送给自己）来执行计算。比如，在一些由事件驱动的环境下，环境会不断地发送消息并且等待诸如鼠标点击和按键等外部事件的响应。

从测试视角的角度，关于消息可以得出如下结论。

(1) 消息有其发送者。发送者可以决定何时发送消息，并且可能会做出错误的决定。

(2) 消息有其接收者。接收者可能接收到非预期的特定消息。当它接收到一条非预期的消息时，接收者可能会对此做出不正确的反应。

(3) 消息可以包含有实际参数。在处理一条消息时，参数能被接收者使用和更改。如果传递的参数是对象，那么在消息被处理前和处理后，对象必须处于正确的状态，而且它们必须实现接收者所期望的接口。

**3. 接口**

接口是行为声明的集合。行为被集中在一起，并通过单个的概念定义一些相关的动作。

接口是由一些规范构成的，这一规范定义了类的一套完整的公共行为。例如，在 C++ 中，通过定义一个抽象的基类，其内部仅包含公有的虚拟方法，可以达到定义接口的目的。

从测试视角的角度，关于接口可以得出如下结论。

(1) 接口封装了操作的说明。这些说明逐步形成了诸如类这种形式的更大分组的规范。如果这一接口包含的行为和类的行为不相符，那么对这一接口的说明就不是令人满意的。

(2) 接口不是孤立的。它与其他的接口和类有一定的关系。一个接口可以指定一个行为的参数类型，使得实现该接口的类可被当作一个参数进行传递。

**4. 类**

类是一组具有相同数据结构和相同操作的对象集合。类的定义包括一组数据属性和在数据上的一组合法操作。在面向对象的程序中，任何被描述的概念最初都必须被声明为类，然后创建由该类定义的对象。创建对象的过程被称作实例化，而创建的结果被称为实例。在一个类中，每个对象都是类的实例，它们都可使用类中提供的函数。一个对象的状态则包

含在它的实例变量中。

**5. 继承**

继承是类之间的一种联系，它允许新类可以在一个已有类的基础上进行定义。一个类对另一个类的依赖，使得已有类的说明和实现可以被复用。这种方法有一个重要的优势，那就是已有类不会被改变，并且对其他任何继承它的新类来说都是一样的。这里所说的新类主要是指子类或派生类，被新类继承的已有类就叫基类。一个基类，以及从这个基类直接或间接继承而得到的派生类，它们共同构成继承层次关系。在继承层次关系中，我们把这个基类叫做“根”。派生类则是从根直接或间接继承过来的。除了根。每个类都有一到多个直接或间接继承的类，每个类也都有若干从它继承过去的派生类。

良好的面向对象软件设计要求，只有在实现一种或一类关系的时候才使用继承，一般是按照类的定义而不是实现来使用它。

从测试视角的角度来看，继承包含以下内容。

（1）继承提供了一种机制，通过这种机制，潜在的错误能够从一个类传递到它的派生类中。测试类的时候要尽早消除这种错误，以免这些错误一并传递到其他类中，产生延迟放大效应。

（2）继承还提供了另一种机制，这种机制能使我们重复使用相同的测试方法。因为子类是从它的父类继承过来的，所以子类也就继承了父类的说明和实现。因此我们也就可以用测试父类的方法对子类进行测试。

**6. 封装**

封装就是把对象的属性和方法结合成一个整体，尽可能掩盖其内部的细节。封装后的对象，只能知道输入和输出，无法了解内部的操作过程，也无法真正了解内部数据的真实状态。这一特征简化了对对象的使用，同时也给测试带来了难度。

**7. 多态**

多态提供了将对象看作是一种或多种类型的能力。编程语言的类型机制可定义用来支持许多不同的类型适应策略。此外，类型的完全匹配可能是最安全的，但多态却支持灵活的设计，同时又易于维护。

多态有几种不同的形式，如参数多态、包含多态、过载多态。参数多态是能够根据一个或多个参数来定义一种类型的能力。包含多态和过载多态在面向对象语言中通常体现在子类与父类的继承关系上。

包含多态是同一个类具有不同表现形式的一种现象。面向对象的编程语言对包含多态的支持（也称为动态绑定），使得参数具有对象替换的能力。为了响应操作请求，当对象的定义与后续对象的定义相符时，对象就可以被相互替换。换句话说，面向对象程序中的发送者能够在用对象作为参数时根据接口进行实现，而不是实现一个完整的类。

包含多态提供了一种强有力的能力，可以任意设计接口并编写代码，而不用考虑其他类的对象是否发送了消息来请求操作。包含多态使得设计和编码比以前更加抽象。实际上，定义一个没有实例的抽象类是非常有用的，而它的子类则有实例存在。抽象类主要是为了定义一个由所有它的派生类支持的接口。

测试视角的角度来看，包含多态具有以下功能。

（1）包含多态允许系统通过增加类来进行扩展，而不是修改已经存在的类。当然在扩展中也会出现出乎预料的交互关系。

（2）包含多态允许任何操作都可以包括一个或多个类型不确定的参数，这样就增加了应该测试的实际参数的种类。

（3）包含多态允许操作指定动态引用返回的响应，因为实际引用的类可能是不正确的，或者不是发送者所期望的。

## 9.2　面向对象软件测试概述

不同于传统的功能模块结构，面向对象软件的结构抛弃了传统的开发模式，其相应的开发设计阶段的需求与验证机制也与以往不一致，因此传统的经典测试方法已经不能满足面向对象软件测试的需要。

### 9.2.1　面向对象软件的特点对测试的影响

相对于传统的软件开发方法，面向对象软件设计是一种全新的开发技术，面向对象程序的封装性、继承性、多态等特性使程序具有较大的灵活性，这给软件测试提出了新的要求，使得面向对象软件的测试更加复杂。面向对象软件开发设计的特征对软件测试的影响可以分为以下 3 个方面。

**1. 信息隐蔽对测试的影响**

类的重要作用之一是信息隐蔽。信息隐蔽是指只让用户知道那些确保用户正确使用一个对象所必需的信息，其他信息对用户来说是隐蔽起来的。封装性是把一个对象的数据和操作数据的方法聚集在一个逻辑单元内，对对象的访问被限制在一个严格定义的界面上。信息隐蔽和封装性把类的具体实现与它的接口相分离，降低了一个类和程序其他各部分之间的依赖，促进了程序的模块化，并在一定程度上简化了类的使用，避免了不合理的操作并能有效地阻止错误的扩散，减轻了维护的工作量。但同时，它们也给程序的测试带来了一定的问题。

信息隐蔽与封装性对面向对象测试带来的主要问题是对象状态难以观察。对象具有一定的状态，所以对于面向对象的软件测试来说，对象的状态是必须考虑的因素。由于信息隐蔽机制的存在，类的内部对外部来说是“不可见的”，它的属性和状态只能通过类自身的方法或函数来获得。这给测试用例的生成带来了一定的困难。为了能够观察到这些属性和状态，以确定软件执行的结果是否正确，测试时往往要在类的定义中增添一些专门的函数，这些函数用于读取对象的状态，以便测试时能考察对象的状态变化。

**2. 继承对测试的影响**

继承性是一种概括对象共性和组织结构的机制，使得面向对象设计更具自然性和直观性，是一种有效的重要手段。子类不但继承了父类中的特征（数据和方法），还可以对继承的特征进行重定义。然而，继承同时也向测试用例设计者提出了新的挑战。即使是彻底复用

的，对每一个新的使用语境也需要重新测试。此外，多重继承增加了需要测试的语境数量，从而使测试进一步复杂化。表现为：

（1）若在子类中重定义了某一继承的方法，即使两个函数完成相同的功能，对重定义的方法也需要重新测试；

（2）若一个类得到了充分的测试，当其被子类继承后，继承的方法在子类的环境中的行为特征需要重新测试；

（3）在多重继承的情况下，从两个不同的父类中所定义的同名（或同构型）的特征中按不同的优先级（或选择方式）在子类中仅选择保留一个版本时，即使所得到的子类的结构与父类相同，但仍然可能需要不同的测试数据集；

（4）若我们对父类中某一方法进行了重定义，仅对该方法自身或其所在的类进行重新测试是不够的，还必须重新测试其他有关的类，如子类和引用类。

总之，继承并未简化测试问题，反而使源代码变得难以理解，一个深层次的底层子类可能只有一两行代码，但却继承了上百种特征。多重继承会显著地增加派生类的复杂程度，导致一些难以发现的隐含错误。这些都在一定程度上增加了测试的难度。

**3. 多态性与动态绑定对测试的影响**

多态性和动态绑定是面向对象方法的关键特性之一。多态的概念是指同一消息可以根据发送消息对象的不同采用多种不同的行为方式。而多态行为操作是根据当前指针引用的对象类型来决定使用正确的方法。动态绑定则决定了一个消息只能在编译或运行时才能够确定它的具体行为。这样，多态性和动态绑定所带来的不确定性，使得传统测试实践中的静态分析方法遇到了不可逾越的障碍。而且它们也增加了系统运行中可能的执行路径，加大了测试用例的数量和选取难度。

## 9.2.2 面向对象软件测试和传统测试的不同

面向对象的测试在许多方面要借鉴传统软件测试方法中可适用的部分，并且在软件开发的具体实践中，也经常混合使用面向对象的开发方法和结构化的开发方法，因此二者存在一定的相通之处。但是，与传统方法相比，面向对象的开发方法又有新的内容和特点，从而导致二者的差异。

**1. 测试的单元不同**

传统软件的基本构成单元为功能模块，每个功能模块一般能独立地完成一个特定的功能。而在面向对象的软件中，基本单元是封装了数据和方法的类和对象。对象是类的实例，有自己的角色，并在系统中承担特定的责任。对象有自己的生存周期和状态，状态可以演变。对象的功能是在信息的触发下，实现对象中若干方法的合成以及与其他对象的合作。对象中的数据和方法是一个有机整体，功能测试的概念不适用于对象的测试。

**2. 系统构成不同**

传统的软件系统是由一个个功能模块通过过程调用关系组合而成的。而在面向对象的系统中，系统的功能体现在对象间的协作上。

相邻的功能可能驻留在不同的对象中，操作序列是由对象间的消息传递决定的。传统

意义上的功能实现不再是靠子功能的调用序列完成的，而是在对象之间合作的基础上完成的。不同对象有自己的不同状态，而且，同一对象在不同的状态下对消息的响应可能完全不同。因此，面向对象的集成测试已不属于功能集成测试。

### 9.2.3　面向对象软件测试分类

由于面向对象的软件开发具有类继承、封装、多态等新特性，其对相应的软件测试提出了新要求，但是面向对象的软件测试与传统面向过程的软件测试仍然具有一定的相似之处。

面向对象的开发模型突破了传统的瀑布模型，将开发分为面向对象分析（OOA）、面向对象设计（OOD）和面向对象编程（OOP）3 个阶段。分析阶段产生整个问题空间的抽象描述，在此基础上，进一步归纳出适用于面向对象编程语言的类和类结构，最后形成代码。由于面向对象的特点，采用这种开发模型能有效地将分析设计的文本或图表代码化，不断适应用户需求的变动。针对这种开发模型，结合传统的测试步骤的划分，可以将面向对象的测试模型划分为：面向对象分析的测试（OOA Test）、面向对象设计的测试（OOD Test）、面向对象编程的测试（OOP Test）、面向对象单元测试（OO Unit Test）、面向对象集成测试（OO Integrate Test）、面向对象系统测试（OO System Test）。

这种划分与面向对象开发的 3 个阶段的对应关系如表 9-1 所示。

**表 9-1　面向对象的软件测试模型划分**

<table>
<tr><th>开 发 阶 段</th><th colspan="3">对应测试活动</th></tr>
<tr><td>面向对象分析</td><td colspan="2">面向对象分析测试</td><td rowspan="4">面向对象系统测试</td></tr>
<tr><td>面向对象设计</td><td>面向对象设计测试</td><td rowspan="3">面向对象集成测试</td></tr>
<tr><td rowspan="2">面向对象编程</td><td>面向对象单元测试</td></tr>
<tr><td>面向对象编程测试</td></tr>
</table>

与传统的测试模型相比，面向对象的测试更关注于对象而不仅仅是完成输入输出的单一功能。因此，针对面向对象软件开发，测试活动可以更早地介入分析和设计阶段，从而更好地配合软件开发过程，减少软件设计缺陷，提高软件质量。

## 9.3　面向对象软件测试模型

如前节所述，面向对象软件的测试可划分为 6 个测试模型。其中，OOA Test 和 OOD Test 是对分析结果和设计结果的测试，主要是对分析设计产生的文本进行，是软件开发前期的关键性测试；OOP Test 主要针对编程风格和程序代码实现进行测试，其主要的测试内容在面向对象单元测试和面向对象集成测试中体现；OO Unit Test 是对程序内部具体单一的功能模块的测试，如果程序是用 C++ 语言实现，主要就是对类成员函数的测试，是进行 OO Integrate Test 的基础；OO Integrate Test 主要对系统内部的相互服务进行测试，如成

员函数间的相互作用,类间的消息传递等;OO System Test 是基于 OO Integrate Test 的最后阶段的测试,主要以用户需求为测试标准,需要借鉴 OOA 或 OOA Test 结果。

尽管上述各阶段的测试构成了一个相互作用的整体,但其测试的主体、方向和方法各有不同。本节将从 OOA、OOD、OOP、单元测试、集成测试、系统测试 6 个方面分别介绍对面向对象软件的测试。

## 9.3.1 面向对象分析测试

传统的面向过程分析是一个功能分解的过程,是把一个系统看成可以分解的功能集合。这种传统的功能分解分析法的着眼点在于一个系统需要什么样的信息处理方法和过程,以过程的抽象来满足系统的需要。而对一个系统而言,与传统分析方法产生的结果相反,其行为是相对稳定的,结构是相对不稳定的,这更充分反映了现实的特性。OOA 的结果是为后面阶段类的选定和实现、层次结构的组织和实现提供平台。因此,对 OOA 的测试,应从以下方面考虑:对认定的对象的测试;对认定的结构的测试;对认定的主题的测试;对定义的属性和实例关联的测试;对定义的服务和消息关联的测试。

面向对象分析阶段的主要工作是需求分析和对类、对象和结构的设计。面向对象分析是把 E-R 图和语义网络模型,即信息造型中的概念,与面向对象程序设计语言中的封装、继承、多态等概念结合在一起而形成的分析方法,最后通常是得到问题空间的图表形式的描述。OOA 直接映射问题空间,全面地将问题空间中实现的功能抽象化。将问题空间中的实例抽象为对象,用对象的结构反映问题空间的复杂实例和复杂关系,用属性和操作表示实例的特性和行为。

在确定需求分析以后,会形成面向对象的分析文档,因此,该阶段的测试主要是针对文档的测试,其考虑包括 6 个方面。

1) 对认定的对象的测试

在面向对象的软件需求分析中,将问题空间中的结构、设备、事件、涉及的人员等实例抽象为对象。在面向对象分析的测试中,应该重点测试以下内容:

(1) 认定的对象是否全面,问题空间中所有涉及的实例是否都反映在认定的抽象对象中;

(2) 认定的对象是否具有多个属性。只有一个属性的对象通常应看成其他对象的属性,而不是抽象为独立的对象;

(3) 对认定为同一对象的实例是否有共同的,区别于其他实例的共同属性;

(4) 对认定为同一对象的实例是否提供或需要相同的服务,如果服务随着不同的实例而变化,认定的对象就需要分解或利用继承性来分类表示;

(5) 如果系统没有必要始终保持对象代表的实例的信息,提供或者得到关于它的服务,认定的对象就无必要;

(6) 认定的对象的名称应该尽量准确、适用。

2) 对认定的结构的测试

认定的结构指的是多种对象的组织方式,用来反映问题空间中的复杂实例和复杂关系。认定的结构分为两种:分类结构和组装结构。分类结构主要描述问题空间中实例的一般与

特殊的关系，而组装结构则着重描述问题空间中实例整体与局部的关系。

(1) 对处于高层的对象，是否在问题空间中含有不同于下一层对象的特殊可能性。

(2) 对处于同一低层的对象，是否能抽象出在现实中有意义的更一般的上层对象。

(3) 对所有认定的对象，是否能在问题空间内向上层抽象出在现实中有意义的对象，高层的对象的特性是否完全体现下层的共性。

(4) 低层的对象是否有高层特性基础上的特殊性。

3) 对认定的组装结构的测试

(1) 整体(对象)和部件(对象)的组装关系是否符合现实的关系。

(2) 整体(对象)的部件(对象)是否在考虑的问题空间中有实际应用。

(3) 整体(对象)中是否遗漏了反映在问题空间中有用的部件(对象)。

(4) 部件(对象)是否能够在问题空间中组装新的有现实意义的整体(对象)。

4) 对认定的主题的测试

主题是在对象和结构的基础上更高一层的抽象，是为了提供面向对象分析结果的可见性。对主题层的测试应该考虑以下方面：

(1) 如果主题个数多且不合理，就要对有较密切属性和服务的主题进行归并；

(2) 主题所反映的一组对象和结构是否具有相同和相近的属性和服务；

(3) 认定的主题是否是对象和结构更高层的抽象，是否便于理解 OOA 结果的概貌；

(4) 主题间的消息联系(抽象)是否代表了主题所反映的对象和结构之间的所有可能关联。

5) 对定义的属性和实例关联的测试

属性是用来描述对象或结构所反映的实例的特性。而实例关联是反映实例集合间的映射关系。对属性和实例关联的测试从如下方面考虑：

(1) 定义的属性是否对相应的对象和分类结构的每个现实实例都适用；

(2) 定义的属性在现实世界是否与这种实例关系密切；

(3) 定义的属性在问题空间是否与这种实例关系密切；

(4) 定义的属性是否能够不依赖于其他属性被独立理解；

(5) 定义的属性在分类结构中的位置是否恰当，低层对象的共有属性是否在上层对象属性中体现；

(6) 在问题空间中每个对象的属性是否定义完整；

(7) 定义的实例关联是否符合现实；

(8) 在问题空间中实例关联是否定义完整。

6) 对定义的服务和消息关联的测试

定义的服务，就是定义的每一种对象和结构在问题空间所要求的行为。由于在问题空间中，实例之间存在必要的通信，因此在面向对象分析中相应需要定义消息关联。对定义的服务和消息关联的测试从如下方面进行：

(1) 对象和结构在问题空间的不同状态是否定义了相应的服务；

(2) 对象或结构所需要的服务是否都定义了相应的消息关联；

(3) 定义的消息关联所指引的服务提供是否正确；

(4) 沿着消息关联执行的线程是否合理，是否符合现实过程；

（5）定义的服务是否重复，是否定义了能够得到的服务。

## 9.3.2 面向对象设计测试

在面向对象设计阶段，主要工作是对面向对象分析的阶段归纳出的类和结构进行详细的设计，从而构造成类库，实现分析结果对问题空间的抽象。由此可见，面向对象设计是对面向对象分析的进一步细化和更高层的抽象。在设计完成以后，同样会形成面向对象的设计文档。因此，该阶段的测试同样是针对文档的测试，其考虑包括 3 个方面。

1）对确认的类的测试

OOD 确认的类可以是 OOA 中认定的对象，也可以是对象所需要的服务的抽象，对象所具有的属性的抽象。确认的类原则上应该尽量保持基础性，这样才便于维护和重用。测试中应从以下方面重点关注确认的类：

（1）是否涵盖了 OOA 中所有认定的对象；

（2）是否能体现 OOA 中定义的属性；

（3）是否能实现 OOA 中定义的服务；

（4）是否对应着一个含义明确的数据抽象；

（5）是否尽可能少地依赖其他类；

（6）类中的方法是否为单用途。

2）对构造的类层次结构的测试

为能充分发挥面向对象的继承共享特性，通常 OOD 的类层次结构基于 OOA 中产生的分类结构的原则来组织，着重体现父类和子类间一般性和特殊性。类层次结构应该能在解空间中，构造实现全部功能的结构框架。测试中应重点注意：

（1）类层次结构是否包含了所有定义的类；

（2）是否能体现 OOA 中所定义的实例关联；

（3）是否能实现 OOA 中所定义的消息关联；

（4）子类是否具有父类没有的新特性；

（5）子类间的共同特性是否完全在父类中得以体现。

3）对类库的支持的测试

对类库的支持虽然也属于类层次结构的组织问题，但其强调的重点是再次软件开发的重用。由于它并不直接影响当前软件的开发和功能实现，因此将其单独提出来测试，也可作为对高质量类层次结构的评估。测试中应重点注意：

（1）一组子类中关于某种含义相同或基本相同的操作，是否有相同的接口；

（2）类中方法功能是否较单一，相应的代码行是否较少；

（3）类的层次结构是否深度大，宽度小。

## 9.3.3 面向对象编程测试

面向对象程序是通过对类的操作来实现软件功能的。更确切地说，是能正确实现功能的类，通过消息传递来协同实现设计要求。因此，在面向对象编程的测试中，需要我们忽略

类功能实现的细则,将测试的目光集中在类功能的实现和相应的面向对象程序风格上,其考虑包括两个方面。

**1. 数据成员是否满足数据封装的要求**

数据封装是数据和与数据有关的操作的集合。检查数据成员是否满足数据封装的要求,基本原则是数据成员是否被外界(数据成员所属的类或子类以外的调用)直接调用。更直观地说,当改变数据成员的结构时,是否影响了类的对外接口,是否会导致相应外界必须改动。但是,有时强制的类型转换会破坏数据的封装特性。

**2. 类是否实现了要求的功能**

类所实现的功能,都是通过类的成员函数执行。在测试类的功能实现时,应该首先保证类成员函数的正确性。单独地看待类的成员函数,与面向过程程序中的函数或过程没有本质的区别,几乎所有传统的单元测试中所使用的方法,都可在面向对象的单元测试中使用。具体的测试方法在面向对象的单元测试中介绍。类函数成员的正确行为只是类能够实现要求功能的基础,类成员函数间的作用和类之间的服务调用是单元测试无法确定的。因此,需要进行面向对象的集成测试。具体的测试方法在面向对象的集成测试中介绍。需要着重声明,测试类的功能,不能仅满足于代码能无错运行或被测试类能提供的功能无错,应该以OOD生成的需求文档为依据,检测类提供的功能是否满足设计的要求、是否有缺陷。如有必要,还应该参照OOA的初始分析结果。

## 9.3.4 面向对象单元测试

面向对象软件的单元概念发生了变化,封装驱动了类和对象的定义。这意味着每个类和类的实例(对象)包装了属性(数据)和操纵这些数据的操作(也称为方法或服务),而不是个体的模块。最小的可测试单位是封装的类或对象。类包含一组不同的操作,并且某些特殊操作可能作为类的一部分特殊功能存在(例如类中的静态函数)。因此,单元测试的意义发生了较大的改变,实际上,面向对象的单元测试可以认为是对类的测试。

面向对象软件的类测试与传统软件的单元测试相对应,但和传统的单元测试不一样。具体地,传统的测试方法只适用于类中方法的测试,不适用于类的整体测试,同时孤立地检查类中方法的正确性不足以保证类在整体上是正确的。面向对象软件的类测试是由封装在类中的操作和类的状态行为所驱动的。因此,类测试不能孤立地测试单个操作,要将操作作为类的一部分,同时要把对象与其状态结合起来,进行对象状态行为的测试。

**1. 测试驱动的实现方式**

由于被测的类一般不可能单独执行,需要实现测试驱动,以实现对目标类的测试。测试驱动的设计本质是通过创建被测类的实例和测试这些实例的行为来测试类。下述例子代码为一个Java矩形Rectangle类,下面以编写Rectangle类测试的驱动为例,介绍常见的几种测试驱动的设计方法。

```
class Rectangle{
      float width;
      float height;
      public Rectangle (float w, float h){
          set(w,h);
      }
      void set(float w,float h){
         if(w>0 && w<50)
             width=w;
         else
             width=1;
         if(h>0 && h<50)
             height=h;
         else
             width=1;
      }
      float perimeter(){
          return (2* (width+height));
      }
}
```

（1）利用 main 函数。利用 main 函数方法实现测试驱动是一个最为简单的方式，直接将每个测试用例写入 main 函数。示例如下：

```
public static void main(String[] args) {
      Rectangle rectangle=new Rectangle (10,20);
      System.out.println("周长为:");
      System.out.println(rectangle.perimeter());
  }
```

（2）嵌入静态方法。在被测类中嵌入静态方法，在静态方法内部实现测试用例的执行，然后调用该静态方法。示例如下：

```
class Rectangle{
         static float width;                          //修改为静态变量
         static float height;                         //修改为静态变量
         public Rectangle (float w, float h){
               set(w,h);
         }
         public static void set(float w,float h){   //修改为静态函数
               if(w>0 && w<50)
                   width=w;
                else
                   width=1;
               if(h>0 && h<50)
                   height=h;
               else
                   width=1;
         }
```

```
        public static float perimeter(){        //修改为静态函数
            return (2*(width+height));
        }
}

//调用静态方法的代码如下:
public class test {
    public static void main(String[] args) {
        Rectangle.set(0, 30);
        System.out.println("周长为:");
        System.out.println(Rectangle.perimeter());
    }
}
```

（3）设计独立测试类。将测试代码从开发代码中完全独立出来，通过独立的测试类处理被测类的实例化和方法，并对结果进行统计。示例如下：

```
public class RectangleTestCase {
    private static int  sum_total=0;
    static BufferedReader buf=new BufferedReader(new InputStreamReader(System.in));
    private static int getTotal(){
        return sum_total;
    }
    private static void setTotal(int total){
        sum_total=total;
    }
    public float setTest(float w,float y){
        Rectangle rectangle=new Rectangle(w,y);
        perimeter=rectangle.perimeter();
        return perimeter;
    }
    public static void testCase(){
        RectangleTestCase test=new RectangleTestCase();
        int total_temp=getTotal();
        setTotal(total_temp+1);
        float width =0,height =0;
        System.out.print("输入长:");
        try {
            height=Float.parseFloat(buf.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.print("输入宽:");
        try {
            width=Float.parseFloat(buf.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(test.setTest(height, width));
```

```
    }
    public static void main(String[] args) {
        String flag=null;
        do{
            System.out.print("输入 1 开始测试,输入其他结束测试");
            try {
                flag=buf.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
            if(flag.equals("1"))
            {
                testCase();
            }
            else
                break;
        }while(flag.equals("1"));
        System.out.println("测试结束");
        System.out.println("总的测试次数为"+getTotal());
    }
}
```

**2. 面向对象单元测试的目标**

(1) 类功能正确性。功能正确性是类测试的基本目标,也是定量分析预期逻辑功能被正确封装实现的可信度。类的功能正确性包括行为正确性和实现正确性。行为正确性用来验证目标类是否正确提供预期设定的服务;实现正确性则主要验证类实现代码是否与规格说明一致。

(2) 类完整性。检查目标类中实现的逻辑过程是单元测试的一个主要方面。完整性可以用来确定目标类是否具备了所需的功能和变量以及类应具有的公共接口。另外,完整性还要求确定一个类的方法是否完整地执行了规定的功能。

(3) 早期测试。为了让软件缺陷尽早被发现并修复,以减少延后效应带来的成本放大效应,应该对类尽可能早的测试,通常在该类与其他类集成之前进行测试。

另外,对一个类单独测试并不能保证该类已经得到充分的测试,还需要引入多个测试用例,以测试类中方法的交互和对象在测试期间的状态。类似传统功能测试,可以把类中方法作为黑盒进行确认,把每个成员函数作为一个单独实体。一个对象在某个时刻的状态等价于这时所有数据成员的聚合状态,类的方法使用各种机制来操作数据成员。一个类的正确性需要验证数据成员是否已代表了对象预期的状态,成员函数是否能够正确地对对象数据进行操作。

## 9.3.5 面向对象集成测试

在面向对象的术语中,集成测试的一个主要目标是确保每个类或组件对象的消息以正确的顺序发送和接收并确保接收消息的外部对象的状态获得预期的影响。如图 9-1 所示,

给出了指令生成序列图，图中箭头即代表类间消息的传递。



**图 9-1　面向对象的序列图示例**

区别于传统软件的功能分解，面向对象软件是通过合成来构造软件的，因而集成是面向对象软件开发中最重要的工作。集成测试主要根据系统中相关类的层次关系，检查类之间的相互作用的正确性，即检查各相关类之间消息连接的合法性、子类的继承与父类的一致性、动态绑定的合法性、类间协同完成系统功能的正确性等。因面向对象软件没有层次的控制结构，传统的自顶向下和自底向上的集成测试策略并不适用于面向对象方法构造的软件，这是因为面向对象的软件的执行实际上是执行一个由信息连接起来的方法序列，而这个方法序列往往是由外部事件驱动的。

此外，在各个方法分别测试之后，每次任选一个方法集成到类中逐步进行测试直至形成一个完整的类的集成策略也未必合适，原因是各个方法之间可能有相互作用，某一方法可能要求对象处于某个特定的状态，而该状态必须由其他方法设置，所以还需要考虑集成的次序问题。

**1. 面向对象的集成测试策略**

面向对象软件的集成测试有两种策略。

1）基于线程的测试

由系统的一个输入事件作为激励，对其触发的一组类进行测试，执行相应的方法/消息处理路径，最后终止于某一输出事件。应用回归测试对已测试过的类集合重新执行一次，以保证加入新类时不会产生意外的结果。

由于基于线程的测试策略比较符合一般的认知规律，并且可以根据需求规格说明的序列图生成消息，并跟踪触发的类及类间消息的传递互动过程，因此，该策略在实际测试中较为常见。以图 9-1 为例，根据基于线程的测试策略，我们可以设计测试用例：向类“遥控指令数据生成主控对象”发出消息——“从内存中读取控制指令信息”，继而引发后续与“控制

指令转换序列”“遥控指令生成”“指令数据反演”“数据库”类之间的交互作用，通过检查类消息传递和交互作用的正确性进行集成测试。

2）基于使用的测试

首先通过测试独立类（是系统中已经测试正确的某类）来开始构造系统，在独立类测试完成后，继续测试下一层继承独立类的类（称为依赖类）。按照继承依赖关系对各类进行测试，直到整个系统被构造完成为止。

**2. 面向对象软件的集成测试过程**

测试过程的第一步是进行静态测试，针对程序的结构，检测其是否符合设计要求。通过专业的测试软件提供的“可逆性工程”的功能，得出源程序的类系统图和函数功能调用关系图。将“可逆性工程”得到的结果与 OOD 的结果相比较，检测程序结构和实现上是否有缺陷，是否达到了 OOP 的设计要求。

另外，类似于 2.1 节，面向对象的集成测试同样也需要进行静态测试，包括代码审查、静态分析、逻辑测试和文档审查等。其中代码审查主要使用基于面向对象的规则集，例如 MISRA-C++：2008、HIC++ 等规则集。表 9-2 列举了部分 MISRA-C++：2008 规则集。详细的规则集可参见相关资料文档。

**表 9-2 MISRA-C++：2008 部分规则集**

| 序号 | 审查项目 | 类别 | MISRA-C++：2008 标识 |
|---|---|---|---|
| 1 | Assignment operation in expression | 强制类 | 05-00-01<br>06-02-01 |
| 2 | No brackets to loop body | 强制类 | 06-03-01 |
| 3 | Unused procedure parameter | 强制类 | 00-01-11<br>00-01-12 |
| 4 | Proc/Program contains Variable (s) declared but not used in code analysed | 强制类 | 00-01-03 |
| 5 | Procedure contains UR data flow anomalies | 强制类 | 08-05-01 |
| 6 | DU data flow anomalies found | 强制类 | 00-01-06<br>00-01-09 |
| 7 | Procedure has more than one exit point | 强制类 | 06-06-05 |
| 8 | Function has no return statement | 强制类 | 08-04-03 |
| 9 | Ellipsis used in procedure parameter list | 强制类 | 08-04-01 |
| 10 | Use of setjmp/longjmp | 强制类 | 17-00-05 |
| … | … | … | … |

测试工具 Testbed 支持多种面向对象的代码规则集，其可以通过扫描源代码很容易地生成规则违反报告表。

测试过程的第二步是动态测试，即根据静态测试得出的函数功能调用关系图或类关系图作为参考，并依据需求规格说明，按照既定的集成测试方法完成测试功能覆盖。

**3. 面向对象的集成测试方法**

具体的面向对象软件集成测试方法主要有以下 6 种。

（1）基于状态的测试。即利用有限状态自动机的相关理论来对面向对象系统进行集成。

（2）基于事件的测试。此方法用同步序列来表示事件之间的关系，依据同步序列来集成各对象，并在此基础上进行测试。

（3）基于形式规约的测试。应用代数规约技术来形式化描述类的行为，进而进行测试。

（4）确定性与可达性测试技术。此方法主要通过执行预定的某个同步序列来检验某个确定性的结果。

（5）变异测试。这是一种错误驱动测试，即针对某类特定程序错误，通过检验测试数据集的排错能力来判断测试的充分性。

（6）基于 UML 的测试。利用面向对象建模语言 UML 的类图、状态图和顺序图等提供面向对象系统的信息来产生测试用例。

## 9.3.6 面向对象系统测试

通过单元测试和集成测试，仅能保证软件开发的功能得以实现，不能确认在实际运行时，是否满足用户的需要，是否存在大量实际使用条件下会被诱发产生错误的隐患。为此，对完成开发的软件必须经过规范的系统测试。换个角度说，开发完成的软件仅仅是实际投入使用系统的一个组成部分，需要测试它与系统其他部分配套运行的表现，以保证在系统各部分协调工作的环境下也能正常工作。

系统测试不仅是确认系统在实际运行时，它是否满足用户的需要，也是对软件开发设计的再确认。因此在进行系统测试时，应该参考 OOA 的结果，对应描述的对象、属性和各种服务。其测试内容包括以下 5 种。

**1. 功能测试**

功能测试是对软件需求规格说明或设计文档中的功能需求逐项进行的测试，以验证其功能是否满足要求。功能测试一般需进行：

（1）用有效值的等价类输入数据值测试；

（2）用无效值的等价类输入数据值测试；

（3）进行每个功能的合法边界值和非法边界值输入的测试；

（4）用一系列真实的数据类型和数据值运行，测试超负荷、饱和及其他“最坏情况”的结果；

（5）在配置项测试时对配置项控制流程的正确性、合理性等进行验证。

**2. 强度测试**

强度测试是强制软件运行在不正常到发生故障的情况下（设计的极限状态到超出极限状态），检验软件可以运行到何种程度的测试。强度测试一般需：

（1）提供最大处理的信息量；

（2）提供数据能力的饱和实验指标；

（3）提供最大存储范围（如常驻内存、缓冲、表格区、临时信息区）；

（4）在能力降级时进行测试；

（5）在人为错误（如寄存器数据跳变、错误的接口）状态下进行软件反应的测试；

（6）通过启动软件过载安全装置（如临界点警报、过载溢出功能、停止输入、取消低速设备等）生成必要条件，进行计算过载的饱和测试；

（7）进行持续一段规定的时间，而且连续不能中断的测试。

### 3. 性能测试

性能测试是对软件需求规格说明或设计文档中规定的性能需求逐项进行的测试，以验证其性能是否满足要求。性能测试一般需：

（1）测试在获得定量结果时程序计算的精确性（处理精度）；

（2）测试其时间特性和实际完成功能的时间（响应时间）；

（3）测试为完成功能所处理的数据量；

（4）测试程序运行所占用的空间；

（5）测试其负荷潜力；

（6）测试配置项各部分的协调性；

（7）在系统测试时测试软件性能和硬件性能的集成；

（8）在系统测试时测试系统对并发事件和并发用户访问的处理能力。

### 4. 安全测试

安全性测试是检验软件中已存在的安全性、安全保密性措施是否有效的测试。测试应尽可能在符合实际使用的条件下进行。安全性测试一般包括：

（1）对安全性关键的软件部件，必须单独测试其安全性需求；

（2）在测试中全面检验防止危险状态措施的有效性和每个危险状态下的反应；

（3）对设计中用于提高安全性的结构、算法、容错、冗余及中断处理等方案，必须进行针对性测试；

（4）对软件处于标准配置下其处理和保护能力的测试；

（5）应进行对异常条件下系统/软件的处理和保护能力的测试（以表明不会因为可能的单个或多个输入错误而导致不安全状态）；

（6）对输入故障模式的测试；

（7）必须包含边界、界外及边界结合部的测试；

（8）对“0”、穿越“0”以及从两个方向趋近于“0”的输入值的测试；

（9）必须包括在最坏情况配置下的最小输入和最大输入数据率的测试；

（10）对安全性关键的操作错误的测试；

（11）对具有防止非法进入软件并保护软件的数据完整性能力的测试；

（12）对双工切换、多机替换的正确性和连续性的测试；

（13）对重要数据的抗非法访问能力的测试。

### 5. 恢复测试

恢复性测试是对有恢复或重置功能的软件的每一类导致恢复或重置的情况逐一进行的测试，以验证其恢复或重置功能。恢复性测试是要证实在克服硬件故障后，系统能否正常地

继续进行工作，且不对系统造成任何损害。恢复性测试一般需进行：

（1）探测错误功能的测试；

（2）能否切换或自动启动备用硬件的测试；

（3）在故障发生时能否保护正在运行的作业和系统状态的测试；

（4）在系统恢复后，能否从最后记录下来的无错误状态开始继续执行作业的测试。

除以上测试外，还可以根据需求进行容量测试、容错性测试、易用性测试以及安装/卸载测试等。

第 10 章 CHAPTER 10

# FPGA 测试

本章首先介绍可编程逻辑器件的基本概念和发展历程，然后介绍可编程逻辑器件测试相关技术及方法。

## 10.1 FPGA 测试概述

### 10.1.1 可编程逻辑器件的基本概念

可编程逻辑器件(programmable logic device，PLD)源于 20 世纪 70 年代，是在专用集成电路(application specific integrated circuit，ASIC)基础上发展起来的一种新型逻辑器件。随着基于计算机的电子设计自动化(electronic device automation，EDA)技术不断发展进步，可编程逻辑器件逐渐成为当今数字机系统设计的主要硬件平台。相比 ASIC 电路，可编程逻辑器件的主要特点是完全由用户通过软件进行编程和配置，从而完成某种特定的功能，且可以反复擦除，从而赋予数字系统设计更多的便利和灵活性，缩短数字系统设计的周期，降低系统开发的成本。

常见的可编程逻辑器件产品主要有可编程只读存储器(programmable read only memory，PROM)、现场可编程逻辑阵列(field programmable logic array，FPLA)、可编程阵列逻辑(programmable array logic，PAL)、可擦除的可编程逻辑阵列(erasable programmable logic array，EPLA)、复杂可编程逻辑器件(complex programmable logic device，CPLD)和现场可编程逻辑门阵列(field programmable gate array，FPGA)等类型。不同类型的可编程逻辑器件的结构、实现方法各不相同。

按照编程工艺的差异，可编程逻辑器件分为 4 类：

(1) 熔丝(fuse)和反熔丝(antifuse)可编程器件；

(2) 可擦除的可编程只读存储器(UEPROM)编程器件；

(3) 电信号可擦除可编程只读存储器(EEPROM)编程器件(如 CPLD)；

(4) SRAM 编程器件(如 FPGA)。

在工艺分类中，前 3 类可编程逻辑器件属于非易失性器件，编程后配置数据将保留在器件芯片中，第 4 类为易失性器件，掉电后，配置数据将会丢失，实际应用中，每次上电后需要重新进行数据配置。

### 10.1.2 硬件描述语言的发展历程

硬件描述语言(hardware description language,HDL)是一种用形式化方法来描述数字电路和系统的语言。数字电路系统开发人员通常采用自顶向下的方法(从抽象到具体)描述设计意图,然后逐层分解为各个模块,逐步细化设计思想。然后,利用 EDA 工具逐层进行仿真验证,把其中需要转换为具体物理电路的模块组合,经由自动综合工具转换为门级电路网表。最后,利用可编程逻辑器件自动布局布线工具将网表转换为具体的物理电路结构。在实现具体的物理电路之前,可以利用硬件描述语言的门级模型(或原语器件)代替具体电路元件。这些门级模型的逻辑功能和延时特性与真实的物理元器件完全一致,因而可以在仿真工具的支持下,在综合前或布局布线前进行逻辑功能和时序的验证,以提高设计效率。这种高层次设计(high level design)被广泛地应用于复杂数字系统的设计中。

硬件描述语言的发展至今已有三十多年的历史,并成功地应用于设计的各个阶段,即建模、仿真、验证和综合。从 20 世纪 80 年代至今已出现了上百种硬件描述语言,极大地促进了数字系统设计自动化的发展。目前,硬件描述语言逐渐向标准化的方向发展,目前主流的硬件描述语言主要有 VHDL、Verilog HDL、System Verilog 和 System C 等,其中 VHDL 和 Verilog HDL 的应用最为广泛。

### 10.1.3 VHDL 语言

VHDL 的英文全称是 VHSIC(very high speed integrated circuit) HDL,于 1983 年由美国国防部发起创建,由 IEEE 进一步发展,并于 1987 年作为 IEEE STD 1076 发布。从此,VHDL 成为硬件描述语言的业界标准之一。此后,各个 EDA 厂商相继推出支持 VHDL 的数字系统开发环境,VHDL 在电子设计领域得到了广泛的应用。

20 世纪 90 年代初,人们发现 VHDL 不仅可以在高层次上描述系统和元器件的行为,作为系统模拟的建模工具,而且可以作为电路系统的设计工具,可以通过 CAD 工具将 VHDL 源码自动转换为文本方式表达的网表文件。进而,EDA 开始推出能够将 VHDL 的部分语句描述转化为具体电路网表的软件工具。IEEE 于 1993 年对 VHDL 进行修订,在更高的抽象层次和系统描述能力上扩展了 VHDL 的内容,发布新版 VHDL 语言标准,即 IEEE Std 1076—1993。目前公布的最新 VHDL 标准版本是 IEEE Std 1076—2002。

VHDL 语言具有很强的电路描述和建模能力,能从多个层次对数字系统进行建模和描述,从而大大简化了硬件设计任务,提高设计效率和可靠性。

VHDL 语言具有与具体硬件电路无关和与设计平台无关的特性,并且具有良好的电路行为描述和系统描述的能力,并在语言易读性和层次化、结构化设计方面,表现出了强大的生命力和应用潜力。利用 VHDL 语言,能够准确简练地表达设计者的原始描述,经过 EDA 工具综合处理,最终生成可供生产的电路描述或版图参数描述的工艺文件。整个过程可以利用 EDA 工具自动完成,使得设计者更加专注于设计的正确性和质量,减少出错概率。

VHDL 语言的设计实体(Design Entity)、程序包(Package)、设计库(Library)为设计的

可重用性提供了强大的支持，有利于设计封装和集成，缩短设计开发时间。

### 10.1.4 Verilog HDL 语言

Verilog HDL 于 1983 年由 GDA(Gateway Design Automation)公司的 Phil Moorby 首创。1984—1985 年，Moorby 设计了名为 Verilog-XL 的仿真器。1986 年，Moorby 提出基于快速门级仿真的 XL 算法。随着 Verilog-XL 算法的成功，Verilog HDL 语言得到迅速发展。

1989 年，Candence 公司收购 GDA 公司，Verilog HDL 语言成为 Candence 公司的私有财产。1990 年 Candence 公司成立 OVI(Open Verilog International)组织负责促进 Verilog HDL 语言的公开和推广。IEEE 于 1995 年制定了 Verilog HDL 语言标准，即 Verilog HDL 1364—1995；于 2001 年发布 Verilog HDL 1364—2001 标准；于 2005 年发布 System Verilog IEEE 1800—2005 标准，这一系列标准使得 Verilog HDL 语言在综合、仿真验证和模块的可充用方面得到大幅提高。

相比其他类型的硬件描述语言，Verilog HDL 具有如下优势。

(1) Verilog HDL 是一种通用型的硬件描述语言，易学易用。Verilog HDL 具有与 C 语言类似的语言风格，因此对于具有 C 语言程序设计经验的开发人员而言，更易掌握和学习；

(2) Verilog HDL 语言允许同一电路模型在不同的抽象层次进行描述。设计者可以从开关、门、RTL(register transfer level)或者行为级进行电路模型的定义。同时，设计者只需学习一种语言用于描述电路的激励信号，并进行层次化设计；

(3) 绝大部分主流 EDA 工具均对 Verilog HDL 提供良好的支持，使得开发者拥有大量可信赖的支持环境；

(4) 编程语言接口(program language interface，PLI)是 Verilog HDL 语言的重要特性之一，使得设计者可以通过编写 C 语言代码访问由 Verilog HDL 语言设计模块的内部结构，从而方便验证工程师使用 PLI 按照自己的需求来配置 Verilog HDL 仿真器。

### 10.1.5 面向可编程逻辑器件的开发过程

随着电子设计技术的不断发展，目前可编程逻辑的设计趋势转向基于计算机的电子设计自动化技术，即 EDA。EDA 技术依赖于功能强大的计算机，在 EDA 工具平台上，以硬件描述语言或原理框图作为系统逻辑描述阶段的设计输入文件，自动完成编译、约简、分割、综合、布局布线和仿真测试，直至实现所要求的数字电路系统。典型的可编程逻辑器件的开发过程如图 10-1 所示。

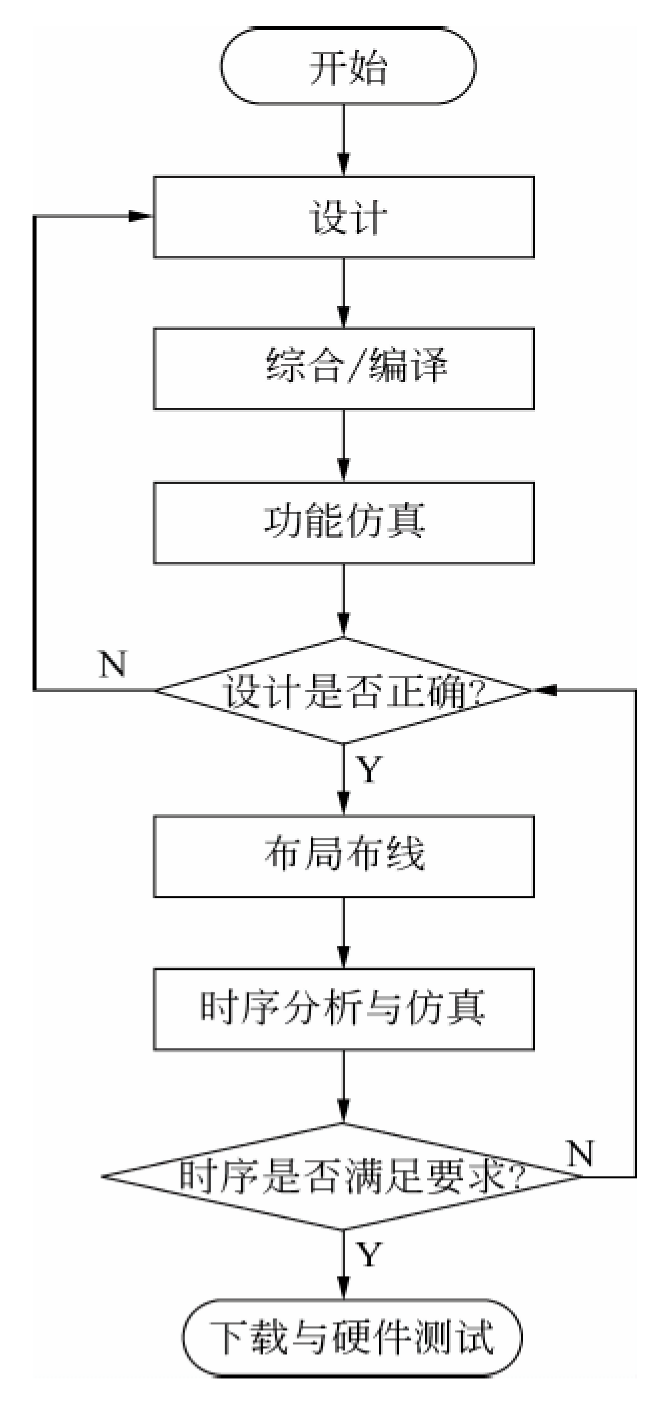



**图 10-1 典型的 FPGA 开发流程**

### 1. 设计

设计是以硬件描述语言或者图形化的方法描述所要实现的电路模型的过程。

1）硬件描述语言输入

使用硬件描述语言，采用自顶向下的层次化方法，以代码的形式描述电路的功能和时序关系，建立可供 EDA 工具识别的电路模型。常用的建模方式有门级建模、数据流建模和行为级建模。该方式便于在不同的层次开展电路的仿真验证工作，且开发模块的可重用性和可移植性更好，且便于大规模复杂设计的协同开发。

2）图形输入

图形通常包括状态图、波形图输入和原理图输入等方法。状态图输入方法即根据电路的控制条件和不同的转换方式，采用绘图的方法利用 EDA 工具的状态图标绘状态图，然后由 EDA 工具将状态变化图综合成电路网表。

波形图输入方法则是将待设计的电路看成是黑盒子，只需在 EDA 工具中配置该黑盒子的输入、输出时序波形图，EDA 工具即可据此完成黑盒子电路的设计。

原理图输入则是在 EDA 工具的图形界面上拖放各类逻辑器件和连线组成特定功能的电路原理图。

图形输入的方式是与 EDA 工具紧密关联的，不同厂商之间的图形难以兼容和互联互通。

### 2. 综合

简单地讲，综合(synthesis)是将抽象的、分散的实体结合成单个或统一的实体。在数字系统设计领域，综合是将用行为和功能层次表达的电路模型转换为低层次的便于实现的便于装备、组合的模块的过程。

设计过程通常从高层次、抽象的行为描述开始，而以最底层的电路结构描述为结束，每一个层次的等效转换都可看作是一次综合：

(1) 从自然语言、图形或高级语言转换到硬件描述语言，是自然语言的综合过程；

(2) 从算法描述转换到寄存器传输级(RTL)描述，及从行为域到结构域的综合，是行为综合；

(3) 从 RTL 级描述转换到逻辑门(触发器等)的表述，即逻辑综合；

(4) 从逻辑门级描述到版图或者转换到配置网表的描述，即结构综合。

综合过程是将软件化的电路模型转换为硬件电路的关键步骤，是从描述到电路的桥梁。整个综合过程就是将设计在 EDA 平台上编辑输入的 HDL 文本、原理图或状态图描述，根据特定的平台、硬件结构组合和约束条件经编译、优化、转换和综合最终获得门级电路甚至更低层次的物理电路描述网表文件。

### 3. 功能仿真

直接对 HDL 或原理图描述的逻辑功能进行测试模拟，以检验其实现的功能是否满足原有的设计要求。该仿真过程不涉及任何具体器件的硬件特性。不经历布局布线，在设计项目编译完成后即可进行功能仿真测试。该仿真的优点在于设计耗时短，且对硬件库和综合器没有要求。

#### 4. 布局布线

布局布线又称结构综合器，其功能是将由综合器产生的网表文件配置于指定的目标器件中，使之产生最终的下载文件。通常，EDA 软件中的综合器可由专业的第三方 EDA 公司提供，而适配器则需由芯片厂商提供，因为适配器的适配对象直接与器件的结构细节相对应。

适配器就是将综合后网表文件针对某一具体目标器件进行逻辑映射操作，包括底层器件配置、逻辑分割、优化、布局布线等操作，适配完成后产生可供进行精确时序仿真的网表文件，以及用于编程下载的二进制文件。

#### 5. 时序分析与仿真

时序分析用于分析不同的信号沿电路中不同路径传播中的时延信息及其接口时序，检查时序是否满足需求。

时序仿真是接近器件真实运行特性的仿真，仿真文件已包含了器件硬件特性参数，仿真精度高。时序仿真的仿真文件必须来自针对特定器件的适配器，综合后所得的网表文件作为 FPGA 适配器的输入文件，产生的仿真网表文件中已包含精确的硬件时延信息。

#### 6. 下载与硬件测试

经过布局布线后生成的下载或配置文件，通过编程器或编程电缆向指定的芯片下载，以便进行硬件调试和验证。

最后，需要将载入了设计的硬件系统进行统一的测试，以便最终验证设计项目在目标系统上的实际工作情况，以排除错误，改进设计。

### 10.1.6 可编程逻辑器件软件与传统软件的不同

首先分析可编程逻辑器件与传统软件系统的差异。通过可编程逻辑器件搭建的系统是以电路为基础的，而传统软件是以通用型硬件平台为基础构建的，两者的主要差异见表 10-1。

**表 10-1 FPGA 系统与传统软件系统的差异性分析**

| | 可编程逻辑器件 | 传统软件系统 |
|---|---|---|
| 编程语言 | 硬件描述语言(VHDL、Verilog、System Verilog 等) | 高级程序设计语言(C、C++ 、Java、C# 等) |
| 开发方式 | 自顶向下、模块化 | 面向过程、面向对象等 |
| 代码执行方式 | 过程体内并行执行 | 顺序执行 |
| 硬件影响 | 硬件性能影响大、与硬件耦合度高 | 硬件影响相对小 |
| 执行结果 | 存在随机性 | 结果具有确定性 |
| 驱动激励 | 信号、时序 | 数据、操作、时间 |
| 应用领域 | 时序要求高 | 对时序要求不高 |
| 运行平台 | 特定芯片，平台耦合度高 | 通用型运行平台 |
| 验证方法 | 时序验证、功能验证、时序分析等 | 软件测试、代码检查、评审等 |

由于可编程逻辑器件开发、设计和应用与传统软件的差异较大，因此传统的软件测试方法难以直接应用于可编程逻辑器件的测试。

### 10.1.7　全过程域的可编程逻辑器件测试框架

与软件系统的测试类似，可编程逻辑器件软件的测试贯穿于可编程逻辑器件的整个开发周期，需要在开发全过程的不同阶段采取相应的技术手段验证设计的正确性。

覆盖全过程域的可编程逻辑器件软件的测试框架如图 10-2 所示。可编程逻辑器件测试技术包括静态测试和动态测试。其中静态测试主要包括文档审查、编码规则检查、代码走查、跨时钟域分析、形式化分析、等效性验证和静态时序分析；而动态测试则包括仿真测试和板级验证，其中仿真测试包括功能覆盖测试、代码覆盖测试、时序仿真测试和软硬协同仿真等。

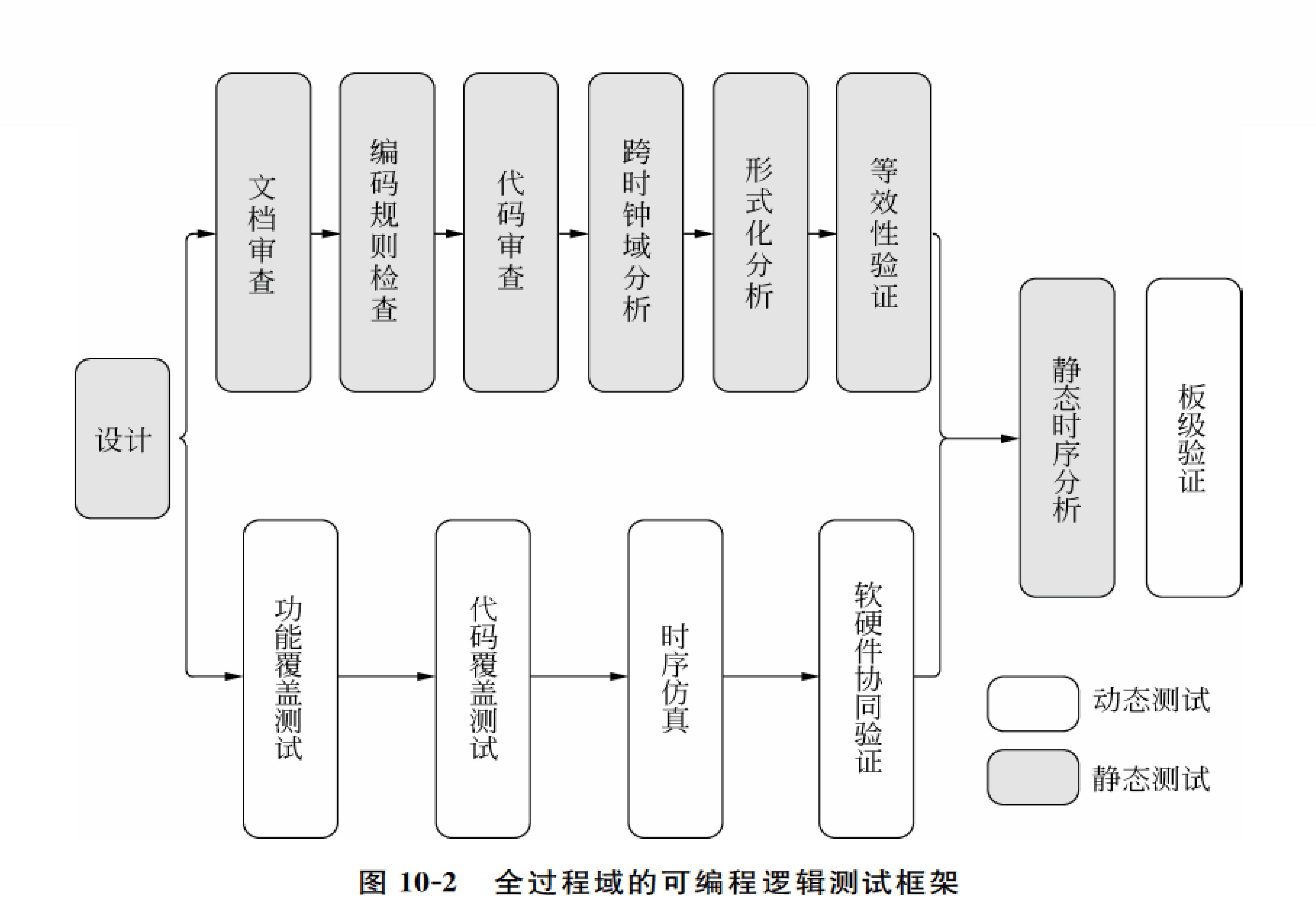



图 10-2　全过程域的可编程逻辑测试框架

## 10.2　静态测试

可编程逻辑器件静态测试的技术手段不需要设计和运行测试用例，即可有针对性地检测出被测设计在特定方面的缺陷，具有快速、全面、高效的特点，是验证被测设计正确性和有效性的重要手段。静态测试的对象可以是开发过程文档、代码、网表文件等，充分利用测试人员的经验和 EDA 工具的自动化分析手段，检查可能存在的设计隐患或需求偏离情况。

## 10.2.1 文档审查

可编程逻辑器件开发过程中产生的技术文档，是沟通实际业务需求、设计和最终实现的桥梁，是开发团队协同工作的依据，为可编程逻辑器件设计的技术维护、升级提供参考。进行文档审查前，应根据被测设计的特点和技术要求制定文档检查单，且文档检查单必须通过技术评审才能作为审查依据。文档审查主要关注以下方面：

(1) 审查文档齐全性；

(2) 审查文档标识和签署的完整性；

(3) 审查文档内容的完备性、准确性、一致性、可追踪性；

(4) 审查文档格式的规范性；

(5) 审查文档内容的正确性。

表 10-2～表 10-5 给出了基本的文档审查单示例。

**表 10-2 软件需求规格说明文档审查表**

| 序号 | 检 查 要 点 | 符合 | 不符合 | 不适用 | 检查情况记录 |
|---|---|---|---|---|---|
| 1 | 是否完整清晰地描述了引用文件，包括引用文档（文件）的文档号、标题、编写单位（或作者）和日期等 | | | | |
| 2 | 是否确切给出了所有在本文档中出现的专用术语和缩略语定义 | | | | |
| 3 | 是否对可编程逻辑器件运行环境、开发环境、软件功能等进行说明 | | | | |
| 4 | 是否完整、清晰、详细地描述了由可编程逻辑器件实现的接口定义、管脚定义及时序要求等需求 | | | | |
| 5 | 是否给出了可编程逻辑器件接口管脚定义说明，是否包含信号名称、信号传输方向、信号数据宽度及信号描述 | | | | |
| 6 | 是否给出了各种接口信号关系、接口信号的时序特性及时序余量要求 | | | | |
| 7 | 是否描述了接口数据元素所需要的特性，例如：数据元素的名称/标识符、数据类型、大小和格式、计量单位（如：米、纳秒）、可能值的范围或枚举（如：0～99）、准确性和精度（有效数字位数）、来源（设置/发送实体）、接收者（使用/接收实体）、时序、频率、容量以及其他约束条件等 | | | | |
| 8 | 是否描述了可编程逻辑器件接口通信特性，如周期/非周期传送、数据传送速率、传输间隔等 | | | | |
| 9 | 是否描述了可编程逻辑器件接口协议特性，如协议的优先级别/层次；合法性检查、错误控制和恢复过程等 | | | | |
| 10 | 是否描述了其他所需要的特性，例如接口实体的物理兼容性（尺寸、公差、负载和接插件的兼容性等）、电压等 | | | | |

续表

| 序号 | 检 查 要 点 | 符合 | 不符合 | 不适用 | 检查情况记录 |
|---|---|---|---|---|---|
| 11 | 是否完整、清晰、详细地描述了由可编程逻辑器件实现的全部功能，包括实现功能、时序要求、算法要求、地址空间分配、软件可编程要求、操作方式、IP 核复用等要求 | | | | |
| 12 | 是否描述可编程逻辑器件的性能需求，如工作时钟频率、时间精度等需求 | | | | |
| 13 | 是否明确提出软件的安全性、可靠性等其他要求，如错误情况处理、上电及复位后接口信号状态等要求 | | | | |
| 14 | 是否描述本文档中的工程需求与“软件系统设计说明”和(或)“软件研制任务书”中的需求的双向追踪关系 | | | | |
| 15 | 文档编制是否规范、内容完整、描述准确一致 | | | | |
| 评价情况 | (描述不符合项性质与统计信息、各不符合项的处理情况、不符合项影响、产品是否具备评审条件、是否可以开展后续工作等) | | | | |

**表 10-3　软件概要设计说明文档审查表**

| 序号 | 检 查 要 点 | 符合 | 不符合 | 不适用 | 检查情况记录 |
|---|---|---|---|---|---|
| 1 | 是否概述了可编程逻辑器件在系统中的作用 | | | | |
| 2 | 是否描述了可编程逻辑器件体系结构(概要)设计；如果设计的全部或部分依赖于系统的状态或方式，是否进行了说明 | | | | |
| 3 | 是否描述了可编程逻辑器件总体结构，说明了模块的划分、模块的功能、系统数据通路等 | | | | |
| 4 | 是否描述了模块间的动态关系，即可编程逻辑器件运行期间模块间的相互作用情况，包括执行控制流程、数据流、状态转换图、时序图、并发执行以及动态行为的其他方面等 | | | | |
| 5 | 是否对接口进行了标识，并通过接口图来描述可编程逻辑器件的相关接口 | | | | |
| 6 | 是否描述了各功能模块的用途、设计决策、开发状态/类型、输入数据、输出数据、资源占用情况等内容 | | | | |
| 7 | 是否描述了各功能模块采用的设计方法 | | | | |
| 8 | 对于算法的设计，是否描述了完成算法由抽象层次过渡到实现层次的转换，是否确定出算法的输入、输出接口，是否描述算法的实现过程 | | | | |
| 9 | 是否描述了各功能模块中采用的可靠性、安全性、可维护性、可测试性和数据安全保密的设计方法 | | | | |
| 10 | 是否建立了软件设计与软件需求的追踪表 | | | | |
| 11 | 文档编写是否规范、内容完整、描述准确一致 | | | | |
| 评价情况 | (描述不符合项性质与统计信息、各不符合项的处理情况、不符合项影响、产品是否具备评审条件、是否可以开展后续工作等) | | | | |

表 10-4 软件详细设计说明文档审查表

| 序号 | 检查要点 | 符合 | 不符合 | 不适用 | 检查情况记录 |
| --- | --- | --- | --- | --- | --- |
| 1 | 是否概述了可编程逻辑器件在系统中的作用 | | | | |
| 2 | 是否标识和详细描述了模块中的各个子模块 | | | | |
| 3 | 是否用控制流图和数据流图描述了子模块间的关系 | | | | |
| 4 | 是否描述了各子模块的设计需求、质量需求和其他约束条件 | | | | |
| 5 | 是否描述了各子模块的设计决策、地址分配、控制方式、接口方式、存储器空间、接口管脚信号详细定义、时序说明、性能指标、设计输入方法、输入数据、输出数据、内部寄存器及寄存器使用说明、接口信号时序特性及时序余量要求、子模块所用到的逻辑、资源占用情况、测试要求等内容 | | | | |
| 6 | 是否描述了各子模块的可靠性、安全性、可测试性设计方法 | | | | |
| 7 | 是否说明了所使用 IP 核的提供厂商、版本信息及 IP 核属性，并描述其逻辑综合及布局布线要求 | | | | |
| 8 | 是否描述了描述版图设计过程 | | | | |
| 9 | 是否准确描述软件详细设计与概要设计的追踪关系 | | | | |
| 10 | 文档编写是否规范、内容完整、描述准确一致 | | | | |
| 评价情况 | （描述不符合项性质与统计信息、各不符合项的处理情况、不符合项影响、产品是否具备评审条件、是否可以开展后续工作等） | | | | |

表 10-5 软件用户手册文档审查表

| 序号 | 检查要点 | 符合 | 不符合 | 不适用 | 检查情况记录 |
| --- | --- | --- | --- | --- | --- |
| 1 | 是否正确地给出所有在本文档中出现的专用术语和缩略语的确切定义 | | | | |
| 2 | 是否准确地描述了可编程逻辑器件的主要功能，包括功能详细说明、使用方式、协议、复位状态、内部寄存器定义、错误情况处理、应用限制等 | | | | |
| 3 | 是否准确地描述了可编程逻辑器件的主要性能指标和特点，如功耗估算、时钟频率等 | | | | |
| 4 | 是否准确地描述了可编程逻辑器件的接口，包括管脚分配、有效电平等 | | | | |
| 5 | 是否准确地描述了可编程逻辑器件软件编程要求，包括可编程寄存器定义、初始化、地址分配、操作方式等 | | | | |
| 6 | 是否准确地描述了可编程逻辑器件供电要求，如单机提供电流的能力、限流设计约束、供电顺序、去耦电容的配置等 | | | | |
| 7 | 是否准确地描述了可编程逻辑器件未用管脚和其他特殊管脚的处理要求 | | | | |
| 8 | 是否准确地描述了可编程逻辑器件上电复位时间要求 | | | | |

续表

| 序号 | 检 查 要 点 | 符合 | 不符合 | 不适用 | 检查情况记录 |
| --- | --- | --- | --- | --- | --- |
| 9 | 是否准确地描述了初始上电或配置过程中可编程逻辑器件的管脚状态及其对单机产品中可编程逻辑器件产品接口处进行隔离设计的要求 | | | | |
| 10 | 是否准确地描述了可编程逻辑器件关键时序信号的相互时序关系及其要求 | | | | |
| 11 | 文档编写是否规范、内容完整、描述准确一致 | | | | |
| 评价情况 | （描述不符合项性质与统计信息、各不符合项的处理情况、不符合项影响、产品是否具备评审条件、是否可以开展后续工作等） | | | | |

## 10.2.2　代码审查

代码走查通常由设计人员、测试人员和任务下达方（或总体人员）一起对设计代码进行人工的检查。设计人员向参加走查的人员讲解设计思路和实现方法，逐行扫描设计代码，走查人员结合自身经验、设计要求，审查设计代码的逻辑正确性和合理性。

对可编程逻辑器件设计代码进行审查前，需要依据被测设计的特点、应用领域、功能需求、工作环境等制定尽可能全面的代码审查单，以保证代码审查的覆盖性、适用性和有效性。

代码审查通常应覆盖以下内容：

（1）审查工程文件的完整性、一致性；

（2）审查代码和设计的一致性；

（3）审查代码执行标准的情况；

（4）审查代码逻辑表达的正确性；

（5）审查代码结构的合理性；

（6）审查代码的可读性。

代码审查单示例如表 10-6 所示。

**表 10-6　代码审查单示例**

| 序号 | 检 查 项 目 |
| --- | --- |
| 1 | 设计中是否有多个时钟，时钟信号的获取和生成是否正确 |
| 2 | 设计中是否有非全局时钟驱动的触发器 |
| 3 | 设计中是否存在跨时钟域信号的处理，处理方式是否合理 |
| 4 | 是否存在异步设计，异步信号的处理是否正确 |
| 5 | 工程中的器件选择是否与目标器件完全相符 |
| 6 | 编程所采取的语言是否符合要求，团队开发中应考虑各个开发团队依据的代码标准是否一致 |
| 7 | 是否存在混合语言编程，兼容性和互操作是否满足要求 |
| 8 | 是否使用 EDA 环境提供的 IP 核，IP 核的例化配置是否正确 |
| 9 | 是否沿用或适配了第三方或其他项目的 IP 核，这些 IP 核是否经过了充分的验证 |

续表

| 序号 | 检 查 项 目 |
|---|---|
| 10 | 设计中是否存在多余的功能模块或语句 |
| 11 | 设计中是否存在未连接/例化的内部或外部接口 |
| 12 | 编码中是否存在不合理的时序逻辑和组合逻辑混用的情况 |
| 13 | 可编程逻辑器件的芯片选型是否合理,温度、电压工作范围是否有足够的余量 |
| 14 | 可编程逻辑器件芯片的处理速度和时钟与外部模块/芯片是否匹配 |
| 15 | 实际应用中是否存在抗辐照需求,如有,则检查所选芯片是否具备所求的抗辐照特性,抗辐照等级是否满足要求 |
| 16 | 芯片外部端口约束是否与芯片选型一致 |
| 17 | 被测设计的片级接口与外部器件接口特性是否一致 |
| 18 | 时钟约束是否全面,时钟过约束设置是否满足要求 |
| 19 | 集成开发环境的分析、综合、布局、布线设置条件是否合理 |
| 20 | 集成开发环境提供的每一个 warning 是否得到确认或验证 |
| 21 | 是否存在有非全局时钟驱动的延迟电路 |
| 22 | 是否有组合逻辑的输出作为异步清零、置数或者时钟输入 |
| 23 | 集成开发环境中 fmax 设置是否大于等于系统主时钟(10%余量) |
| 24 | 时序分析报告中是否有失败路径 |
| 25 | 三种工况下的时序余量能否满足要求 |
| 26 | 是否有要求某寄存器初始状态必须为“0”或“1”而却未明确初始化 |
| 27 | 是否使用了芯片提供的全局复位信号,内部子模块的复位信号传递是否正确 |
| 28 | 复位信号是否有效,系统是否对所有信号赋予了明确的初始值 |
| 29 | 是否存在使用了乘法器但却未调用器件本身提供的专用乘法器 |
| 30 | 是否有乘法器工作于高速的数据率 |
| 31 | 由语言生成的模块中参数(parameter)设置是否正确 |
| 32 | 由语言编辑的模块是否确认过生成的 RTL |
| 33 | 是否有亚稳态的设计 |
| 34 | 是否有注释与设计不符 |
| 35 | 修改模块后是否更新了注释和版本信息 |
| 36 | 资源利用率是否接近 100% |
| 37 | 由 RTL 代码生成网表的综合选项设置是否合理 |
| 38 | 生成布局布线文件的编译选项是否合理 |
| 39 | 生成可供下载的二进制文件的选项设定是否合理 |
| 40 | 有关状态机编码的选项是否正确 |

代码审查是一种以人工为主的技术手段。代码审查工作开始前应详细地策划审查进度(通常精确到模块级),制定详细的工作计划,建立代码审查小组。代码审查小组通常应包含项目技术领导、上层系统的开发人员、本项目外的经验丰富的开发人员、项目开发人员。为保证审查的效果,需要建立相对独立和封闭的工作环境。项目审查结束后应当形成正式的代码审查报告,作为开发人员进行设计修改和改进的依据。开发人员应针对代码审查报告中的每一个问题给出处置意见,并修改相应的设计或文档。最后,代码审查小组应确认所有问题均得到了合理的处置。

一般地,代码审查应尽早进行,以便尽早地发现结构性、全局性的问题,降低设计修改的成本。

## 10.2.3　编码规则检查

编码规则检查针对测试输入的设计源代码(VHDL 或 Verilog),检查其遵循编码规则、规范的情况。编码规则检查的内容包括设计风格、可读性、时钟使用、可综合性、可测试性和模块化等方面的综合性检查,有助于提高代码的规范性,缩短其在 EDA 工具上的时间开销,提高后期仿真和验证的效率、代码的安全性、可靠性和可维护性等。

所谓的编码规则来自于业界广泛认可的、规范的 FPGA 设计和开发的规则集,常用的规则集主要有 Do-254 规则集、Starc 规则集等。

编码规则检查只能检查代码实现过程中不符合规范、不合理的地方,但无法检查代码中可能存在的逻辑错误或冗余。

目前编码规则检查的支持工具主要有 Alint、Leda、SpyGlass、HDL Designer 等,各类工具所支持的内容略有不同,侧重点也有差异。实际项目中应根据被测方的要求,合理地选择和使用的规则集和检查工具。

进行编码规则检查时,应根据被测设计的特点,选择相应的规则集,并对规则集进行适当地裁剪,以提高编码规则检查的针对性和有效性。

表 10-7 为利用 SpyGlass 编码规则检查工具扫描某 FPGA 项目 RTL 代码得到的结果,该设计编码中存在多处违反编码规则的代码段。

**表 10-7　某 FPGA 项目编码规则检查结果**

| 序号 | 等级 | 文件 | 行号 | 违反规则 | 问 题 描 述 |
|---|---|---|---|---|---|
| 1 | 错误 | timer. v | 167 | W19 | Constant 18'd337990 will be truncated |
| 2 | 警告 | fenpin. v | 31 | SYNTH_89 | Initial Assignment at Declaration for (count) is ignored by synthesis |
| 3 | 警告 | Top. v | 101 | RegInputOutput-ML | Port 'fpga_input_inf1' is not registered [Hierarchy: 'Top'] |
| 4 | 警告 | filter. v | 48 | RegInputOutput-ML | Port 'Bufin' is not registered [Hierarchy: 'filter'] |
| 5 | 错误 | ram_read. v | 48 | UndrivenInTerm-ML | Detected undriven input terminal raddr[6: 0] |

续表

| 序号 | 等级 | 文件 | 行号 | 违反规则 | 问 题 描 述 |
|---|---|---|---|---|---|
| 6 | 错误 | ram_write. v | 48 | UndrivenInTerm-ML | Detected undriven input terminal waddr[6：0] |
| 7 | 错误 | control. v | 580 | W122 | The signal/variable 'mas' (or some of its bits) read in the block is not in the sensitivity |
| 8 | 警告 | set. v | 97 | W336 | Blocking assignment used inside a 'FlipFlop' inferred sequential block |

## 10.2.4 跨时钟域分析

随着 EDA 技术的不断发展进步，可编程逻辑器件在数字电路设计中的应用越来越广泛，所实现的功能越来越复杂，设计的规模越来越庞大。在采用同步时序控制的可编程逻辑器件设计中，所有的触发器均在同一个时钟节拍下发生状态翻转，这类设计简单、高效，且有利于后端的综合、优化和布局布线，但单时钟只适用小规模、功能简单的系统设计。复杂的大规模设计难以在单个时钟域中实现所有的功能，并满足时序要求，因而不可避免地引入多个时钟，各个时钟在频率和相位上存在差异，需要处理复杂的跨时钟域(Clock Domain Crossing)问题。

**1. 跨时钟域问题的发生机理**

在可编程逻辑器件设计中，亚稳态是当信号在异步电路中或是无关的时钟域之间传输时导致数字系统失效的一种现象。由于触发器在时钟沿的驱动下进行数据的采集和更新，如果输入数据的更新不能满足建立时间和保持时间的要求，触发器将不能按照预期的目标实现翻转。若输入数据与采样时钟发生“沿打沿”的情况，双稳态电路就不能在规定的时间内进入稳态。一旦双稳态电路出现亚稳态，则电路的输出便缺乏可预测性，输出电压处于非法的电平值，电路输出稳定的时间不可预知，且存在振荡的风险。

当信号从一个时钟域传送到另一个时钟域时，出现在新的、不相关时钟域的信号是异步信号。异步信号的输入可以在当前时钟的任意时刻发生状态翻转。如果信号的状态翻转无法满足后端触发器的建立-保持时间要求时，可能出现状态竞争、数据丢失、状态无效等现象，产生亚稳态。

以 D 触发器为例说明亚稳态的产生原因。图 10-3 展示了输入数据在相对时钟沿的不同时刻发生翻转时，D 触发器的输出情况。建立时间 $T_{su}$是指有效时钟沿到来之前，输入数据需要保持状态稳定的时间，以保证采样时间；保持时间 $T_{hd}$是指输入数据在有效时钟沿之后需要保持的时间，以保证输出数据的稳定。输入数据 $a$ 的建立时间和保持时间都能都满足，数据输出稳定有效；输入数据 $b$ 不能满足保持时间，输入数据 $c$ 不能满足建立时间，从而导致 D 触发器的输出呈现不稳定的状态或者在高低电平之间振荡，即进入亚稳态。

触发器是一个双稳态电路，其输出具有两个稳定态：“0”和“1”。图 10-4 所示为 D 触发器的双稳态锁存电路逻辑图。Q 和 $\overline{Q}$ 是通过反相器首尾相接构成反馈电路，正常情况下，Q 和 $\overline{Q}$ 是互补的，发生状态迁移时，两者的变化趋势相反，其变化曲线如图 10-5 所示。

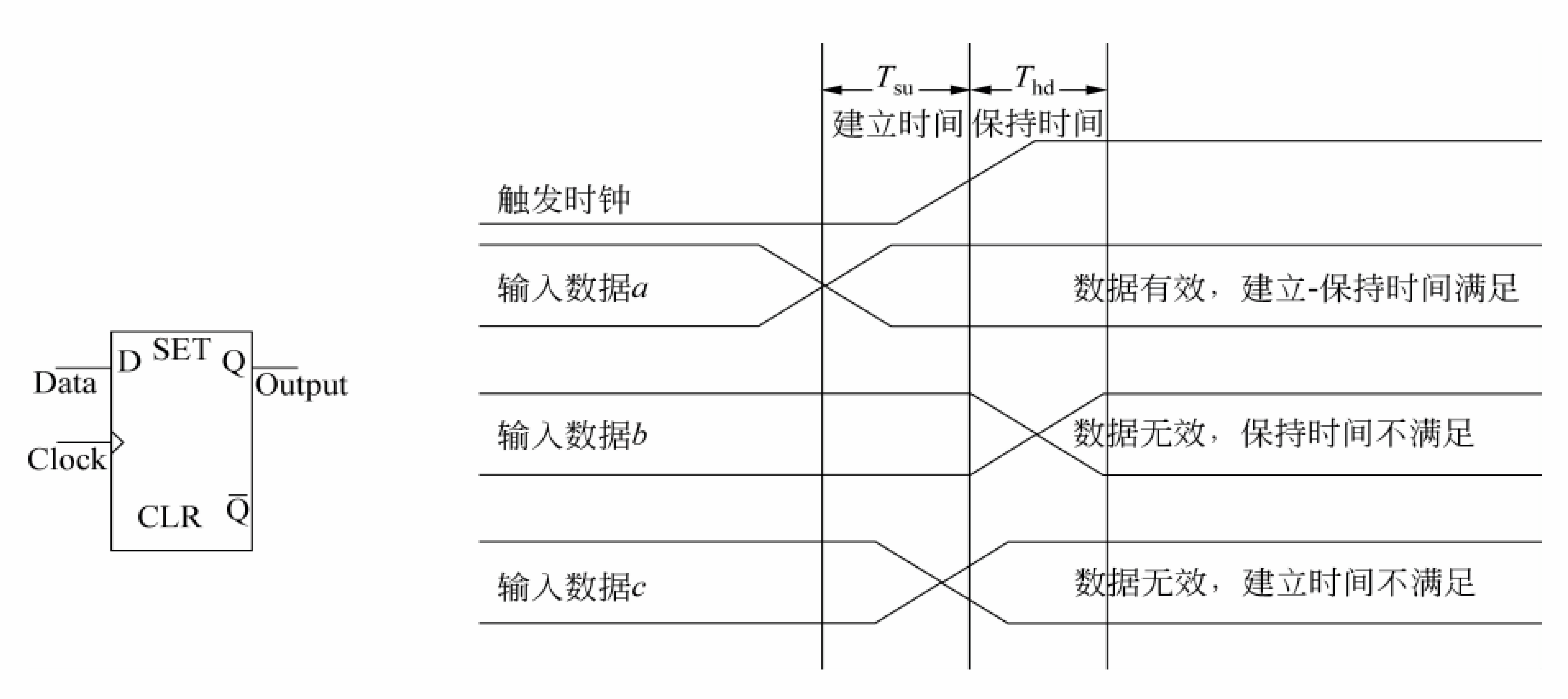



图 10-3　D 触发器亚稳态产生示意图

Q 和 $\overline{Q}$ 端电压的变化需要一定的时间，亚稳态窗口的大小取决于器件的物理特性，其大小远小于稳态的持续时间，但如果将 Q 和 $\overline{Q}$ 的电压置于亚稳态点的电压值上，双稳反馈电路将可能停留在该状态。电路的输出状态发生迁移（从一个稳态过渡到另一稳态）时，需要足够的能量激励，如果所需的能量激励不能满足，则有可能出现停留在亚稳态点的状态，随机噪声的影响会导致状态迁移偏离预期。

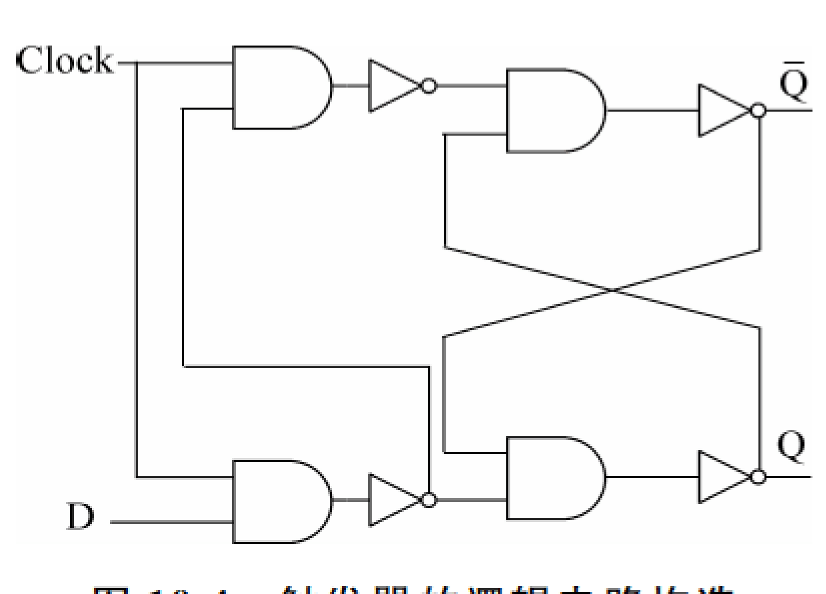



图 10-4　触发器的逻辑电路构造

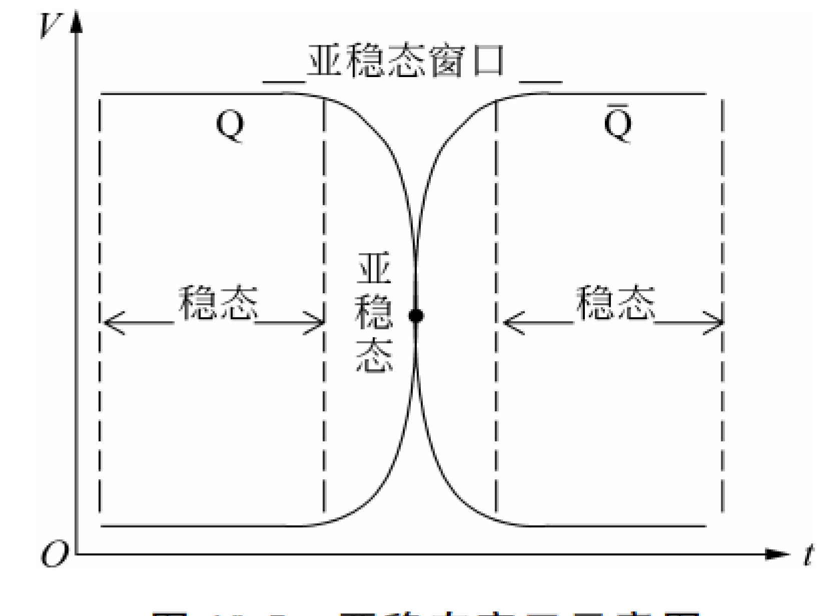



图 10-5　亚稳态窗口示意图

**2. 亚稳态的影响**

由跨时钟域问题导致的亚稳态是影响 FPGA 可靠性和可用性的重要因素，其对数字系统的影响具有如下特点。

（1）亚稳态是双稳态电路所固有的属性，是不可避免的，是依概率发生的。为提高系统的可靠性，只能尽可能地减少亚稳态的发生。

（2）亚稳态具有偶发性和温度敏感性，很难在测试和验证中完全暴露，因此在设计阶段应尽力避免。

（3）亚稳态导致后端逻辑发生混乱，对于多扇出的模块影响更严重。当前设计模块处于亚稳态时，输出值在时间上具有不确定性，不同扇出的延迟不完全相同，因此后端逻辑得到的值存在差异，进而导致亚稳态的影响存在次级传播。错误的逻辑输出在系统内扩展，进而导致整个系统的功能与预期存在较大差异。

**3. 跨时钟域问题的检测**

亚稳态问题难以通过仿真和静态时序分析等传统的验证手段进行检测，需要运用专门的跨时钟域分析工具检查设计中的跨时钟域信号是否进行了有效地同步处理，从而分析跨时钟域信号传播的正确性及芯片工作的可靠性，降低电路失效的风险。

利用相关的跨时钟域分析工具时，重点分析可能存在跨时钟域风险的设计缺陷。

1）信号进行跨时钟域传输时，缺少同步电路支持

当触发器在当前时钟信号的驱动下采集来自其他时钟域的信号时，如果恰好采集到信号的变化边缘而非稳定状态，则容易引发亚稳态。如图 10-6 所示，信号 A 由触发器 F1 输出，F1 的驱动时钟沿为 Clk_A 的上升沿，触发器 F2 的驱动时钟为 Clk_B 的上升沿。如果在 Clk_B 的上升沿，A 信号发生变化，尚未达到稳定状态时，F2 的采样输出具有不确定性，从而引发亚稳态，此时信号 B 的值可能为逻辑“0”(F2 输出端电压接近 Vl)或者逻辑“1”(F2 输出端电压接近 Vh)。

2）快时钟域到慢时钟域信号保持时间不足

当一个快时钟域的脉冲信号发生改变时，如果改变之后信号状态保持的时间小于目标触发器的时钟周期，则该脉冲信号有可能无法被慢时钟域的触发器采集到，进而导致设计失效。

如图 10-7 所示，Clk_A 的时钟频率高于 Clk_B，当信号 A 发生脉冲型变化，且保持时间为 1 个 Clk_A 时钟周期时，由于 Clk_B 相对于 Clk_A 的时钟相位关系不固定，A 的变化存在保持时间不能满足 F2 采样的要求，即 F2 触发器在 Clk_B 的上升沿可能无法采集到信号 A 的脉冲，进而电路忽略信号 A 脉冲，可能导致电路逻辑功能偏离预期。

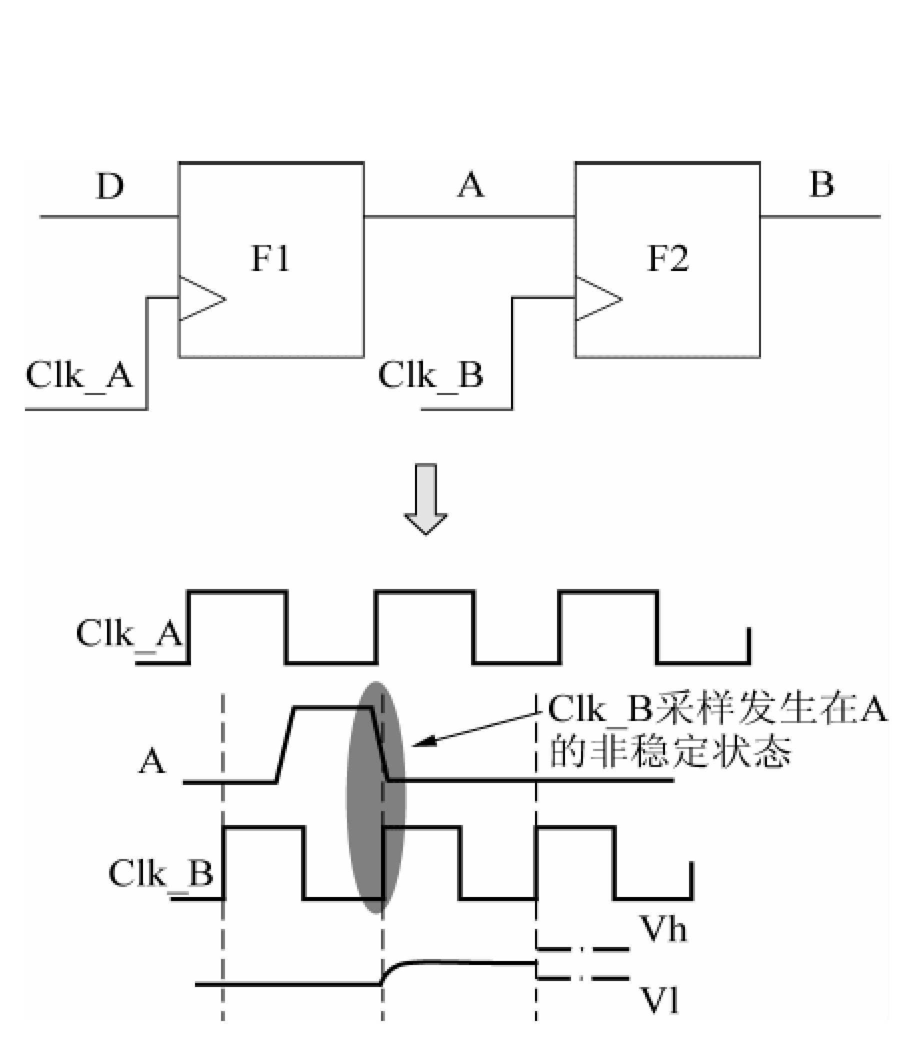



**图 10-6 缺少同步电路的信号传输**

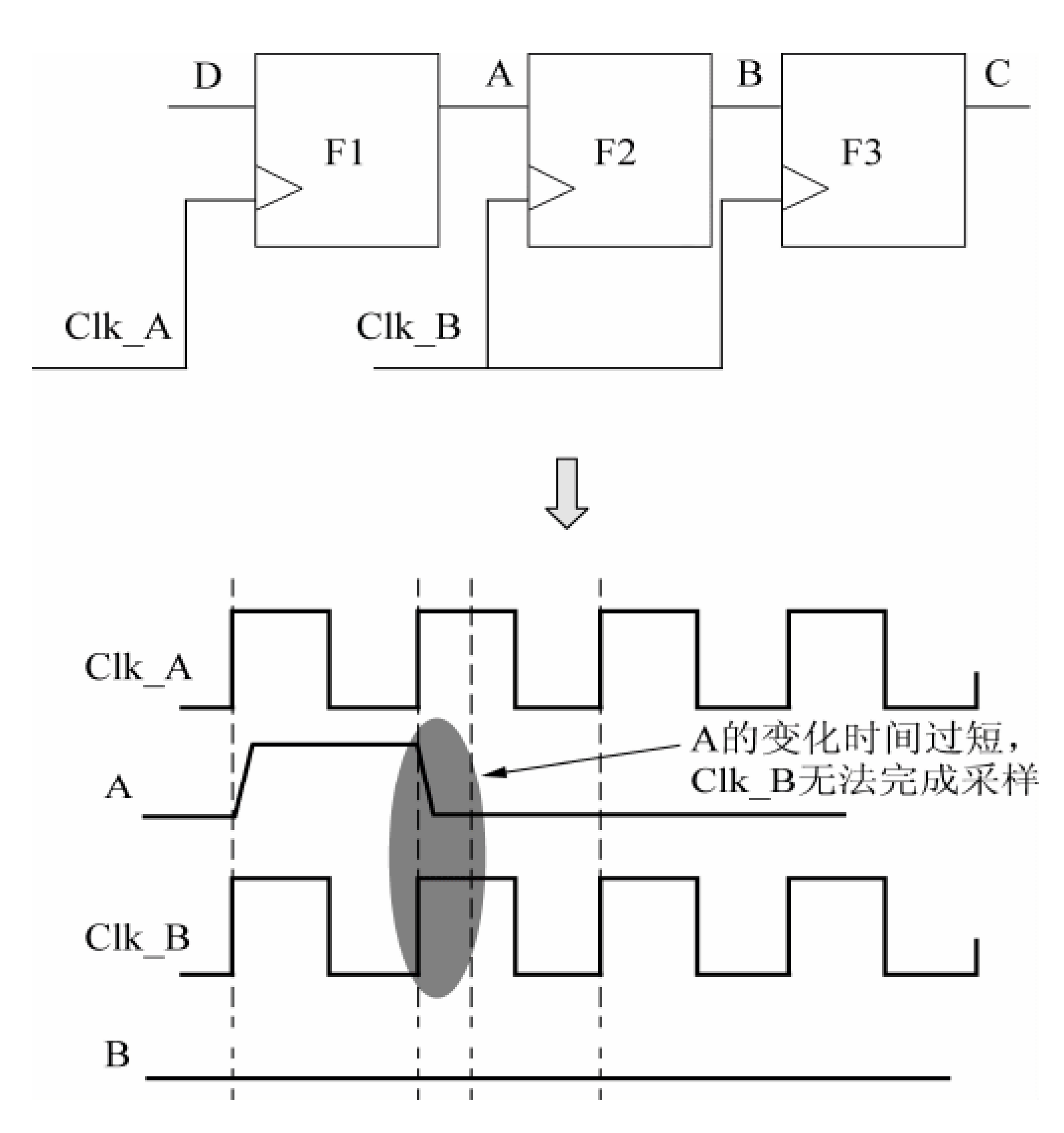



**图 10-7 快时钟域到慢时钟域的脉冲信号传递失效**

3）多比特信号相关性和冲突风险

多比特信号通常为地址或数据总线信号不同数据位之间需要保持一致性，如果不同数

据位信号的传输路径延迟不一致，后端进行信号重组时，数据位之间的一致性有可能发生丢失，从而导致电路功能不正确。

如图 10-8 所示，X、Y 分别为总线的某两位，两路信号保持信号值翻转的一致性，由于 X1 和 Y1 经过的电路延迟不同，所有经过一级触发器同步后的 X3、Y3 失去了一致性，进而末端触发器在时钟 Clk_B 的驱动下采集 X4、Y4 时，采集到的信号已经失去一致性，从而导致信号重组后的结果不正确。

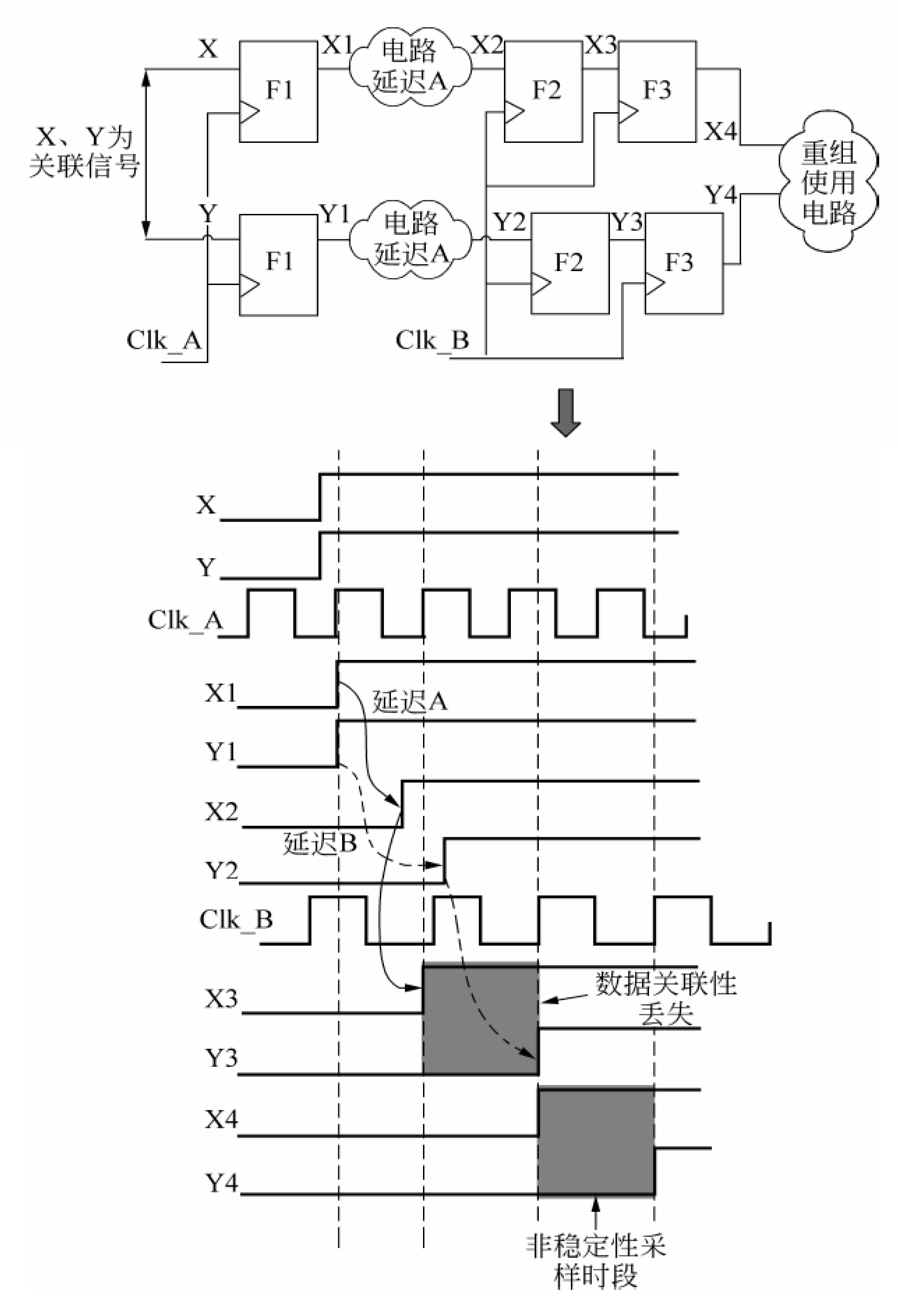



**图 10-8　多比特信号重组时的冲突**

4）FIFO 同步电路中空、满标识生成不正确

FIFO 机制常用于进行跨时钟域的数据传输，完成不同速率的，特别是存在突发性的数据接口处理。典型的双口 RAM 结构的 FIFO 同步控制电路如图 10-9 所示。如设计中使用了 FIFO，则应重点检查空、满信号的生成是否正确，以避免出现写入一个满 FIFO 或读取一个空 FIFO 的情况。同时，为保证地址字段各个信号线之间的数据一致性，应采用格雷码等安全编码以避免可能出现的地址不稳定的状态。

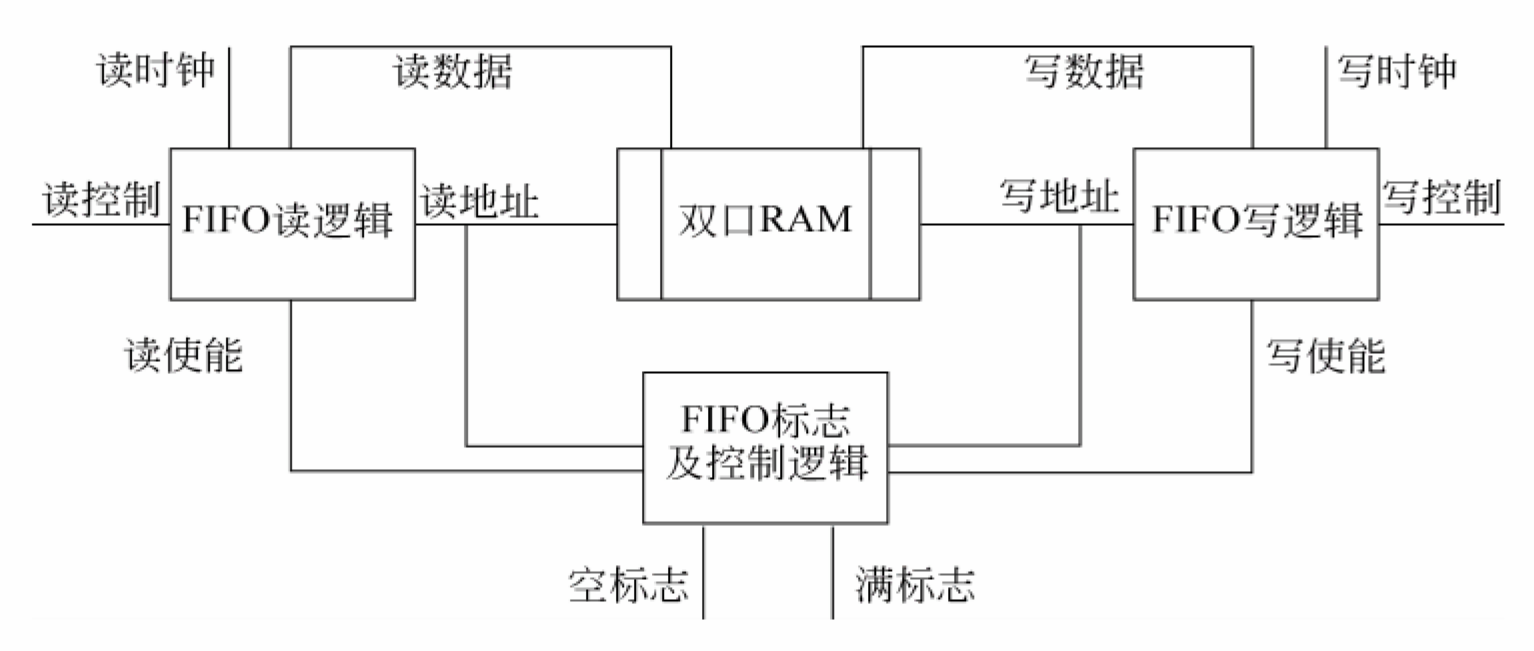



**图 10-9 FIFO 同步机构**

**4. 跨时钟域分析工具**

目前应用比较广泛的跨时钟域分析工具主要有 Questa CDC 和 SpyGlass CDC。这两个工具功能类似，而 SpyGlass CDC 在复杂握手和 FIFO 同步方面有所侧重，且对于大规模设计的检测效率更高。

跨时钟域分析工具 SpyGlass CDC 能够自动地对 FPGA 设计进行模块级和芯片级的扫描，分析时钟信号的敏感沿，检查可能触发亚稳态的跨时钟域问题。该工具能够直接扫描混合语言编写的 HDL 设计文件集，自动分析时钟传播树，检查复杂握手信号和 FIFO 同步设计是否满足设计方法学的要求。将检测结果按照违规严重等级分别以信息、警告或错误显示，并以图形化界面显示出现问题的时钟信号传播路径，并给出建议性的修正策略。

表 10-8 为利用 SpyGlass CDC 对某大型 FPGA 设计进行跨时钟域分析后所得的结果，该设计中有多个存在跨时钟域风险的设计模块。

**表 10-8 某 FPGA 项目跨时钟域分析结果**

| 序号 | 错误等级 | 文件 | 行号 | 违反规则 | 问题分析 |
|---|---|---|---|---|---|
| 1 | 错误 | Oper. v | 39 | Ac_unsync01 | zero_d 的置位是由时钟 GC10M 驱动的，而 operation_8620 的输入端 plo 来自组合逻辑 pla&plb，该组合逻辑中 pla 和 plb 分别由不同的时钟驱动，未进行有效的时序同步操作，存在亚稳态风险 |
| 2 | 错误 | comp. v | 15 | Ac_unsync01 | ageb 是属于 DCM_60MHz. clkfx_out 时钟域的，而 compal 的 b 输入端 diff 是在 GC10M 输出的，两者未进行有效的同步操作，存在亚稳态风险 |
| 3 | 错误 | inf. v | 94 | Ac_unsync02 | thresh 的输出是由时钟 DCM_60MHz 驱动的，而 thresh 信号的更新输入 low_add 是由 Threshold 经过组合逻辑处理得到的，Threshold 源于 threshold_rf，该信号由时钟 GC10M 驱动，两者未进行有效的同步操作处理 |

续表

| 序号 | 错误等级 | 文件 | 行号 | 违反规则 | 问题分析 |
| --- | --- | --- | --- | --- | --- |
| 4 | 警告 | ram_read. v | 32 | Ac_unsync02 | pulse_cnt 是由时钟 CLKDV_BUF 驱动的，而 pulse_cn 的计数控制端是由 reset_n&pulse 的组合逻辑得出的，reset_n 是由时钟 GC10M 驱动的，两者未进行有效的同步操作 |
| 5 | 警告 | ctrl. v | 1272 | Clock_converge01 | dac_0 模块中对 p2s_e 的操作使用采用时钟的下降沿，而其他部分对 p2s_state 的使用均采用上升沿 |
| 6 | 警告 | clock_trans. v | 14 | Reset_sync02 | CLKDV_BUF'时钟域的信号 sign 由 GC10M 时钟域生成的复位信号 reset_n_fifo 执行复位操作，而后者未进行同步操作 |

## 10.2.5 等效性验证

**1. 等效性验证的目的**

如前所述，通常的可编程逻辑设计流程中，开发过程中，从硬件描述语言到可供下载的芯片需要经过综合、映射、布局布线等多个阶段，并呈现出不同的体现形式。开发人员直接面向硬件描述语言、图形或虚拟元器件组成的代码或设计图。而进行信号处理的实体是固化在可编程逻辑芯片中的数字化电路，而中间需要生成综合后网表、时序仿真网表、布局布线后网表等形式。

如何保证整个开发流程中所呈现的不同实体之间是等效的？如何确认各个实体均没有偏离用户的原始意图？为此，可以采取两种策略。

第一种策略是利用仿真手段，在开发中的每一个阶段，针对每一个层次的实体执行同一类的仿真测试，如果两个实体在所有仿真中的运行结果均一致，则说明两个实体是等效的。这种方法需要的仿真时间和计算资源开销极大，仿真手段相对比较低效。

第二种策略是基于形式化方法的逻辑等效性验证，形式化方法是采用理论证明的手段证明各个不同阶段或层次的实体在逻辑上是等效的。这种方法能够保证验证的充分性和完备性，效率远高于基于仿真的验证手段。

利用等效性验证和静态时序分析相结合的方式取代部分时序仿真，确认布局布线后的逻辑实体与经过功能仿真验证的逻辑是否等价，无需测试向量即可快速、全面地完成时序验证工作，以提高测试的效率。

**2. 逻辑等效性验证的作用**

按照逻辑等效性验证的待验证实体的不同，可以划分为 3 种验证形式。

1) RTL 到 RTL 的等效性验证

RTL 级的逻辑等效性验证主要有两种使用方法。

(1) RTL 代码设计修改后的回归测试。如果设计人员对 RTL 代码进行了修改和优化，为了保证所进行的 RTL 更动不影响原有设计的相关功能，可以进行 RTL 级的等效性

验证手段，而不需要运行相应的仿真测试，提高回归测试的效率。

(2) 设计语言迁移有效性的验证。由于实际工程的需要，往往会对相同的设计模型(或IP核)采取不同的硬件描述语言(VHDL、Verilog HDL、System C等)进行实现，验证同一模型的不同实现版本的一致性需要采取逻辑等效性验证手段。

2) RTL到门级的等效性验证

这种验证手段用于比较RTL级设计和门级网表的一致性。主要有以下两种使用方法。

(1) 双重设计的等效性验证。当设计团队采取两种形式对设计进行了描述，其中RTL级的描述用系统仿真和模拟，门级描述用于流片。利用RTL到门级的等效性验证确保两个独立设计在功能上的一致性，从而保证最终实现未偏离预期。

(2) 综合后的回归测试。RTL级的代码具有平台无关性，不同的EDA综合工具的处理基本一致，但会存在差异性。为了验证经过不同EDA综合工具处理后生成的网表在逻辑上与原有的RTL设计一致，确保不同的综合工具未造成不期望的逻辑更动，则可以使用RTL到门级的等效性验证手段，而无需采用运行大量的仿真测试。

3) 门级到门级的等效性验证

门级到门级的等效性验证主要用于难以在门级(时序)运行所有仿真测试的大型设计的验证中，主要用于以下4个方面。

(1) 综合后的时序和功耗改进。对于大规模的复杂设计，综合工具常常不能够迅速、有效地满足所设定的时序和功耗约束，此时需要设计者对各种综合选项、参数进行调整或者直接对综合后的网表进行手动修改，而要验证这些调整和修改未影响所要实现的功能则需要进行门级到门级的等效性验证。

(2) 布局布线后验证。综合后的网表需要经过布局布线工具的处理后才能满足各种时序约束并降低路由/布线拥塞，为了验证布局布线后的网表文件未影响原有的功能，对于小型设计可以采用时序仿真的方法运行所有的功能仿真测试以确保功能复合要求，而对于复杂的大规模设计，时序仿真几乎无法做到全覆盖，因此需要只需验证综合后网表与布局布线后的网表的等效性即可。

(3) 测试逻辑插入后验证。如果在原始的门级设计中插入一个测试逻辑，采取等效性验证方法验证插入测试逻辑后的设计在功能上与原有设计在系统层面上一致的，则说明该测试逻辑未影响原有设计、是可用的。

(4) 回归测试。采用等效性验证方法验证不同版本的门级设计版本在功能上与参考设计版本的一致性，从缩短不同版本迭代过程中回归测试的工作量，提高回归测试的效率。

**3. 等效性验证的工具支持**

常用的等效性检查工具为Formalpro和Formality。这两个工具采用形式验证技术判断一个设计的两个版本在功能上是否等效。这两个工具提供流程化的图形界面和调试功能，检测设计中的错误并将其隔离。功能测试通过后，测试人员可以利用该工具，直接将综合后的门级网表或布局布线后带延时信息的网表与功能仿真后的代码进行等效性验证，确保其未偏离原始的设计意图。

**4. 等效性验证的基本流程**

可编程逻辑器件等效性验证的基本流程如图10-10。在输入设计中A一般作为参考设

计，而 B 作为待验证的设计。输入文件用于设定各类参数，包括黑盒子生命、约束、匹配规则等。库文件是和 EDA 平台及工具相关的、综合、编译和布局布线需要的各类文件。在编译阶段，等效性验证工具将输入设计转化成内部的数据结构并确定各类参考比较点，匹配阶段则建立参考设计与待验证设计各类寄存器、端口和连线的对应关系，分析阶段则综合各类匹配信息，评估 A、B 之间是否等价，如不等价则分析并定位不等价的比较点。最后生成各类报告文件和检查结果。

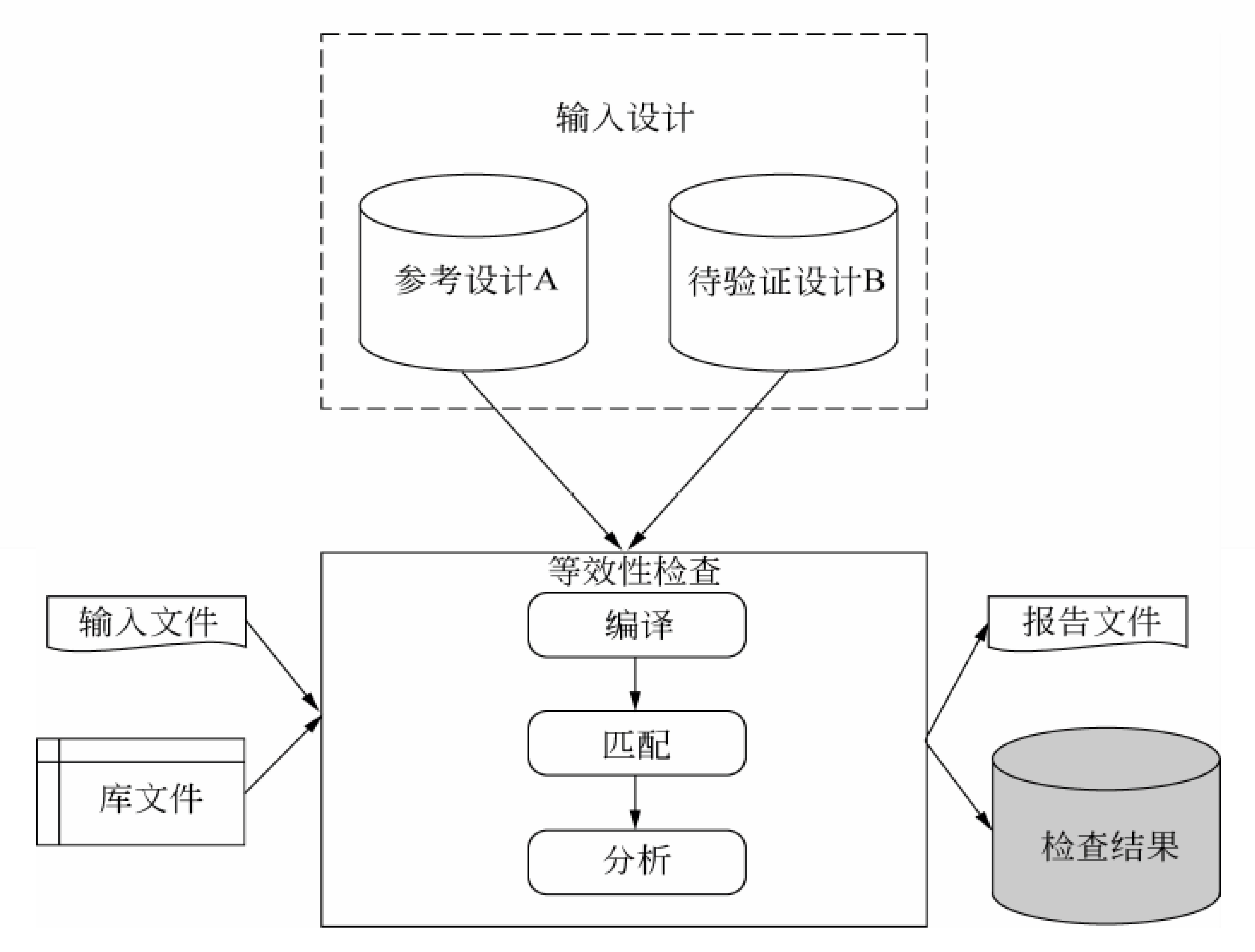



图 10-10　等效性验证基本流程

利用 Formalpro 工具对某小型 FPGA 设计进行了逻辑等效性验证的结果，比较的两端分别为 RTL 设计代码和布局布线后的网表文件。Formalpro 工具的分析报告如下：

```
Comparison Summary
==================
  Total number of comparison points:          60
  ---------------------------------------------------------

  Number of Equivalent comparison points:      58
  ---------------------------------------------------------
    Merged multiply driven net      0
    Solved combinational Cycle      0
    Fed by unmatched net            0

  Number of Different comparison points:       2
  ---------------------------------------------------------
    Plain differences                    2
    Fed by unmatched net                 0
    Ignorable differences                0
    Assertion failures                   0
```

```
  Number of Removed comparison points:        0
  --------------------------------------------------------
    Fed by a multiply driven net       0
    Fed by a combinational cycle       0
    Fed by a floating net              0
    Fed by an unmatched net            0

  Number of Unsolved comparison points:       0
  --------------------------------------------------------
--------------------------
-Designs are DIFFERENT -
--------------------------
```

经过对 RTL 设计和综合后网表的仔细比对，发现存在两个不等价点是由于布局布线后产生了两个多余的反相器导致的。

**5. 等效性验证的关注点**

在执行两个设计实体的等效性验证时，主要的步骤是进行参考点匹配，并对存在的不等价点进行分析，确定其影响域。对于大型的设计往往需要手动编写匹配规则，并采取调试手段分析未匹配的逻辑节点和设计模块。在此过程中，应重点关注以下方面。

（1）寄存器的赋值与合并。综合器或者布局布线器为了满足特定的约束要求，需要对各个设计节点的扇入、扇出度进行调整，从而导致寄存器的复制或者合并。这种不匹配通常存在于不同层次的设计实体的配置中，如 RTL 到门级或者布局布线后验证中。

（2）状态机编码。常用的状态机编码主要有二进制编码、One-Hot 编码和格雷码。设计者采用 HDL 语言编写状态机之后，在综合或布局布线阶段，EDA 工具会根据用户配置采取相应的状态机编码。此时，各个状态值的编码方式有可能与设计者最初设计的编码不一致。因此，在进行等效性验证时应注意检查综合或布局布线后的状态机编码是否合理，状态机相关的各个寄存器的对应关系是否正确，是否满足要求。

（3）状态机的转换。设计者以编程手段描述状态机时，依据的是利用实际需求构造的有限状态机的状态转移图。如果状态转移图比较复杂，则该状态机本身具有一定的优化空间，在综合阶段 EDA 工具将按照既定的规则进行状态裁剪、合并，从而使得处理后的状态机与原有状态机不一致。因此，包含复杂状态机的设计进行等效性验证时，需要格外关注状态机的优化是否正确、合理。另一方面，进行安全状态机的设计，例如设计包含三模冗余的抗干扰状态机设计时，如果不同模式的状态机设计不完备、不正确，在综合阶段，EDA 工具有可能将等价的状态机合并，从而导致三模冗余设计失效。对于复杂状态机三模冗余失效的情况，往往需要运行大量门级仿真或时序仿真才能发现，而采用等效性验证方法则可以较为便捷地发现此缺陷。

（4）锁存器的异常引入。在利用硬件描述语言进行模型描述时，编码中有可能发生 if else 分支不完备、switch case 语句未设计 default 分支、子模块使能端置位不正确（存在常有效/无效使能模块）、子模块端口链接不全等失误，从而导致综合后锁存器的引入，可能导致不匹配点的出现。进行等效性验证时应检查各个锁存器的出现是否符合设计意图，以便修正设计错误。

### 10.2.6　静态时序分析

**1. 目的**

静态时序分析(static timing analysis)用于分析综合或布局布线后的网表是否满足时序要求。时序特性是否符合要求是 FPGA 设计的重要验证内容。时序特性可以通过动态时序仿真进行测试，但该方式具有仿真速度慢、无法验证到所有传播路径等缺点。而静态时序分析方法则能够提取电路的所有时序路径，具有不依赖于激励、可穷尽所有路径、运行速度快等优点，非常适合大规模逻辑设计的验证，是 FPGA 静态测试中不可或缺的测试手段。

**2. 作用与要求**

静态时序分析无需用向量去激活某个路径，对所有的时序路径进行错误分析，能处理百万门级的设计，分析速度比时序仿真快几个数量级。在同步逻辑情况下，可以达到 100%的时序路径覆盖。静态时序分析的目的是找出隐藏的时序问题，根据时序分析结果优化逻辑或约束条件，使设计达到时序闭合(收敛)。静态时序分析可以识别的时序故障数要比仿真多得多，包括建立/保持时间和恢复/移除检查、时钟脉冲宽度、不受约束的逻辑通道。

静态时序分析主要包含如下内容：

(1) 验证被测设计的所有路径的时序需求能够满足；

(2) 分析被测设计的所有约束路径的建立时间和保持时间在当前的工作时钟下能够满足；

(3) 分析被测设计的工作频率是否满足性能要求；

(4) 分析并检查是否存在未约束路径中存在可能导致系统性能瓶颈的关键时序路径；

(5) 分析时钟质量，检查时钟的抖动、偏移、占空比失真等特性对于目标设计的影响程度；

(6) 确定分配引脚的特性，在静态时序分析中，可以通过约束可编程逻辑器件的片级 I/O 引脚所支持的接口标准、接口速率及其他电气特性，然后分析各种接口特性对于时序特征的影响程度。

静态时序分析至少需要覆盖 3 种工况下的时序特性，包括最大工况(温度最高，电压最低)、最小工况(温度最低，电压最高)和典型工况(典型温度和典型电压)。电压和温度是作为约束条件作用于静态时序分析过程的。一般地，温度和电压的变化范围以相应芯片手册中的标称规格为准，也可以根据实际芯片的工作环境适应范围确定。

**3. 原理**

以 2 级触发器级联的简单电路为例说明静态时序分析(static timing analysis)的原理。图 10-11 为一个两级触发器电路。

为保证分析结果的可靠性，所有的分析都是在最恶劣的情况下进行。针对每个触发器，都应满足如下时序方程：

$$t_{SU} \leqslant T_c - (t_{PROP_MAX} + t_{CO_MAX}) + t_{SKEW} \tag{10-1}$$

$$t_{HD} \leqslant t_{PROP_MIN} + t_{CO_MIN} - t_{SKEW} \tag{10-2}$$

式中，

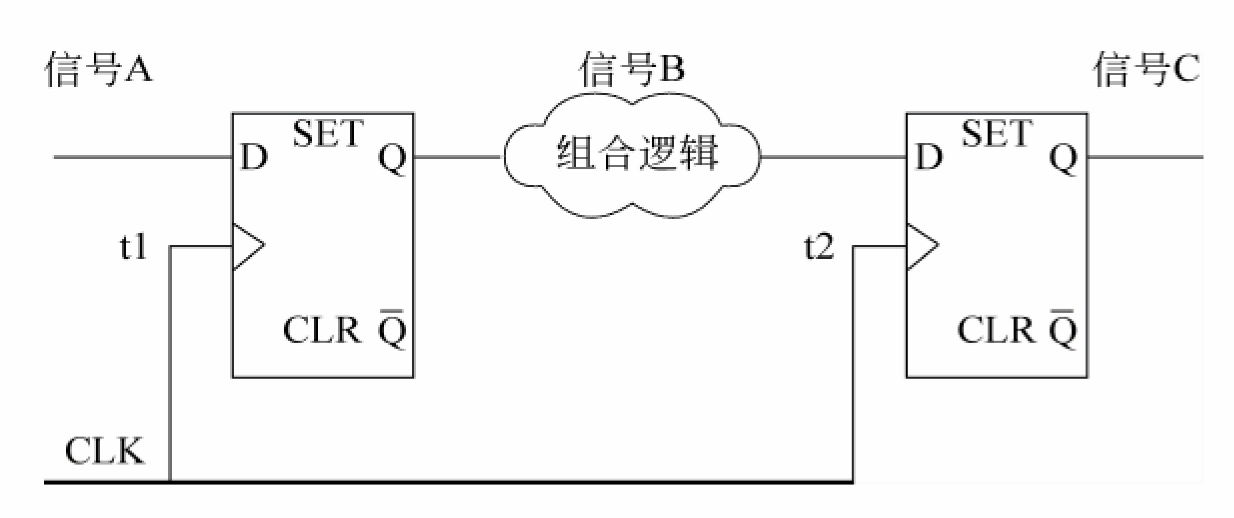



图 10-11 两级触发器电路

$t_{SU}$：触发器要求的建立时间；

$t_{HD}$：触发器要求的保持时间；

$T_c$：系统驱动时钟周期；

$t_{PROP_MAX}$：传输过程最大延迟（包括组合电路延迟和走线延迟）；

$t_{PROP_MIN}$：传输过程最小延迟（包括组合电路延迟和走线延迟）；

$t_{CO_MAX}$：触发器最大传播延迟，即从时钟触发器沿开始到数据稳定的出现在输出端的最大延迟；

$t_{CO_MIN}$：触发器最小传播延迟，即从时钟触发器沿开始到数据稳定的出现在输出端的最小延迟；

$t_{SKEW}$：时钟偏斜，此处 $t_{SKEW}=t_2-t_1$，$t_2$ 和 $t_1$ 分别表示时钟沿到达两个触发器的时间。

对于公式(10-1)，首先，假定时钟 $t_{SKEW}=0$，当前时钟触发沿结束后，第一级触发器的 D 端数据最长需经过 $t_{PROP_MAX}+t_{CO_MAX}$才能到达第二级触发器的 D 端，然后等待 $t_{SU}$，才能允许下一个时钟沿的到来。所以 $t_{PROP_MAX}+t_{CO_MAX}+t_{SU}$是这条路径上的最小时钟周期，从而限制该路径上的最快时钟速度。

对于公式(10-2)，当前时钟触发沿结束后，第一级触发器的 D 端数据最短需要经过 $t_{PROP_MIN}+t_{CO_MIN}$时间到达第二级触发器的 D 端，这也是第二级触发器 D 端旧数据能够保持稳定的时间，所以要求 $t_{PROP_MIN}+t_{CO_MIN}>t_{HD}$。

由于 $t_{CO}$和触发器之间的组合逻辑延迟，第一级触发器的输出信号不会马上反映到第二级触发器的输入端，组合逻辑的延时越大，则 $t_{SKEW}$的值越大，整个系统的建立时间余量逐渐变大，但保持时间余量逐渐减小。若传播延迟过大，导致违背触发器的建立保持时间指标，触发器就可能出现亚稳态的现象。而在异步传输中，由于两个时钟之间不存在相关性，第一个触发器的输出可能在后端触发器时钟沿的任何位置出现，需采取同步措施，以保证系统工作的稳定。

静态时序分析工具，依据设计人员设定的时钟和管脚约束，根据布线结果计算各个时序路径上的延迟信息，分析时钟树的传播，并计算各个模块所需要的建立保持时间能否满足。

**4. 方法**

静态时序分析是基于时序路径的，它将设计逐层分解成 4 种主要的时序路径，如图 10-12 所示。每条时序路径包含 1 个起点和 1 个终点，时序路径的起点只能是设计的基本输入端口或者内部寄存器的时钟输入端，终点则只能是内部寄存器的数据输入端或设计的基本输出端口。

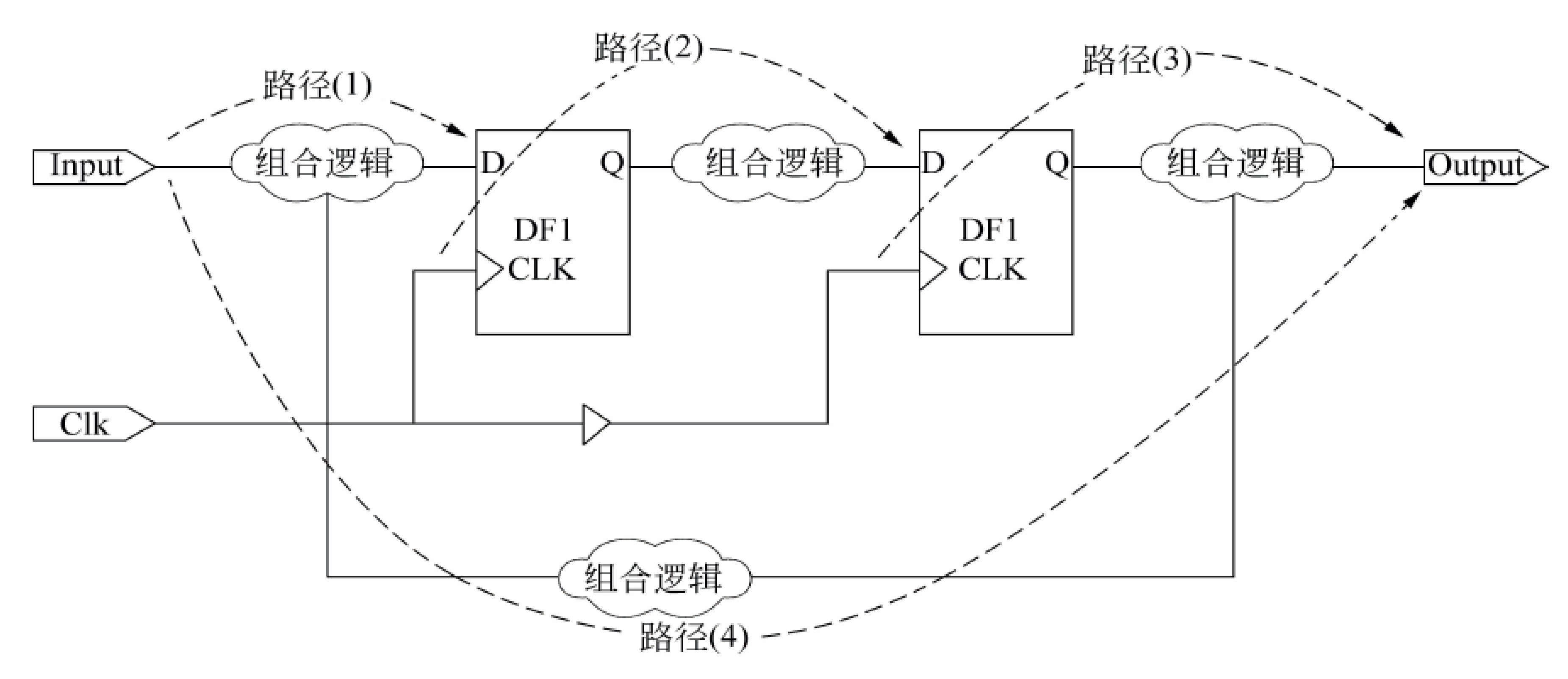



图 10-12 静态时序分析的基本路径

静态时序分析主要包含 4 类基本时序电路：

（1）从输入端口到触发器的数据 D 端；

（2）从触发器的时钟 CLK 端到触发器的 D 端；

（3）从触发器的时钟 CLK 端到输出端口；

（4）从输入端口到输出端口。

通过计算各个时序路径上数据信号的到达时间和要求时间的差值，按照式（10-1）和式（10-2）计算各个时钟节点的建立时间和保持时间，以判定是否存在时序违规或者错误。数据到达的时间 $T_{req}$ 是指数据从起点到终点经过的所有器件和连线延迟时间之和。数据要求时间 $T_{arv}$ 是指根据约束条件计算出的从起点到达终点的理论时间，默认的参考值是 1 个时钟周期。如果数据能够在要求的时间内到达终点，则该路径是符合设计规则的。

时间裕量 Slack 的计算公式为

$$\text{Slack} = T_{req} - T_{arv} \tag{10-3}$$

分析过程中，需要利用工具穷尽计算所有路径上的 Slack，值为正则代表符合要求，值为负则表示不满足。Slack 值为负的路径是影响这个设计时序指标和性能的关键路径，需要进一步优化和改进。

**5. 工具支持**

目前，业界主流的静态时序分析工具主要分成两类。

（1）可编程逻辑器件开发 IDE 标配的工具。不同的芯片提供商分别提供适合自家芯片平台的静态时序分析工具，这类工具通常与开发 IDE 集成在一起，如 Xilinx 公司的 ISE 开发环境包含的静态时序分析工具 Timing Analyzer、Altera 公司的 Quartus Ⅱ 开发环境包含的 TimeQuest Timing Analyzer 以及 Actel 公司的 Libero 开发环境包含的 SmartTime 等。

（2）通用型的静态时序分析工具如 PrimeTime、TimeCraft 等。该这类工具只能够适用于多种开发平台的设计产品，需要开发方提供分析所需要的标准延时文件 SDF 以及与芯片相关的各类库文件。

下面为利用 PrimeTime 工具对某个 FPGA 设计的 TS_C10M 时钟的传播路径进行静态时序分析的结果。工具给出了各个时钟节点的 Slack 值，以及各个路径的延迟信息，据此

可以计算路径的时序余量。如果 Slack 值为负则时序不满足，认定为时序违规，需要进一步处理。

```
================================================================
Timing constraint: TS_C10M =PERIOD TIMEGRP "C10M" 100 ns HIGH 50%;
  12251 paths analyzed, 1102 endpoints analyzed, 0 failing endpoints
  0 timing errors detected. (0 setup errors, 0 hold errors, 0 component switching
limit errors)
  Minimum period is   5.046ns.
Paths for end point set_los/set_8620/sclk (SLICE_X10Y43.G1), 1 path
---------------------------------------------------------------
Slack (setup path):     47.971ns (requirement - (data path - clock path skew +
                        uncertainty))
   Source:              set_los/set_8620/counter_4 (FF)
   Destination:         set_los/set_8620/sclk (FF)
   Requirement:         50.000ns
   Data Path Delay:     1.909ns (Levels of Logic =1)
   Clock Path Skew:     -0.120ns (0.398 -0.518)
   Source Clock:        GC10M rising at 0.000ns
   Destination Clock:   GC10M falling at 50.000ns
   Clock Uncertainty:   0.000ns

   Maximum Data Path: set_los/set_8620/counter_4 to set_los/set_8620/sclk
        Location      Delay type  Delay(ns)        Physical Resource
                                                  Logical Resource(s)
    SLICE_X16Y37.XQ     Tcko        0.360     set_los/set_8620/counter<4>
set_los/set_8620/counter_4
     SLICE_X10Y43.G1      net (fanout=5)  1.287set_los/set_8620/counter<4>
     SLICE_X10Y43.CLK     Tgck         0.262   set_los/set_8620/sclk
set_los/set_8620/sclk_mux00001
set_los/set_8620/sclk
     ---------------------------------------------------
     Total                                     1.909ns (0.622ns logic, 1.287ns route)
                                                     (32.6% logic, 67.4% route)
   Paths for end point set_los/set_8880/sclk (SLICE_X14Y4.G3), 1 path
-----------------------------------------------------------
Slack (setup path):     48.197ns (requirement - (data path - clock path skew +
                        uncertainty))
   Source:              set_los/set_8880/counter_4 (FF)
   Destination:         set_los/set_8880/sclk (FF)
   Requirement:         50.000ns
   Data Path Delay:     1.725ns (Levels of Logic =1)
   Clock Path Skew:     -0.078ns (0.424 -0.502)
   Source Clock:        GC10M rising at 0.000ns
   Destination Clock:   GC10M falling at 50.000ns
   Clock Uncertainty:   0.000ns
   Maximum Data Path: set_los/set_8880/counter_4 to set_los/set_8880/sclk
     Location             Delay type        Delay(ns)  Physical Resource
                                                     Logical Resource(s)
```

```
    SLICE_X23Y3.XQ      Tcko        0.340   set_los/set_8880/counter<4>
set_los/set_8880/counter_4
    SLICE_X14Y4.G3      net (fanout=5)  1.123set_los/set_8880/counter<4>
    SLICE_X14Y4.CLK     Tgck                 0.262   set_los/set_8880/sclk
set_los/set_8880/sclk_mux00001
set_los/set_8880/sclk
    Total                             1.725ns (0.602ns logic, 1.123ns route)
                                          (34.9% logic, 65.1% route)
  Paths for end point set_los/set_8620/sclk (SLICE_X10Y43.G2), 1 path
---------------------------------------------------------------------
Slack (setup path): 48.276ns (requirement - (data path - clock path skew + uncertainty))
  Source:              set_los/set_8620/senb (FF)
  Destination:         set_los/set_8620/sclk (FF)
  Requirement:         50.000ns
  Data Path Delay:     1.612ns (Levels of Logic = 1)
  Clock Path Skew:     -0.112ns (0.398 - 0.510)
  Source Clock:        GC10M rising at 0.000ns
  Destination Clock:   GC10M falling at 50.000ns
  Clock Uncertainty:   0.000ns
    Maximum Data Path: set_los/set_8620/senb to set_los/set_8620/sclk
    Location             Delay type         Delay(ns)  Physical Resource
                                                     Logical Resource(s)
      SLICE_X17Y43.XQ      Tcko         0.340   set_los/set_8620/senb
set_los/set_8620/senb
    SLICE_X10Y43.G2      net (fanout=4)     1.010   set_los/set_8620/senb
    SLICE_X10Y43.CLK     Tgck      0.262   set_los/set_8620/sclk
set_los/set_8620/sclk_mux00001
set_los/set_8620/sclk
    -------------------------------------------------------------
    Total                             1.612ns (0.602ns logic, 1.010ns route)
                                            (37.3% logic, 62.7% route)
=========================================================================
```

## 10.3　仿真测试

按照测试对象所对应的实体不同，可以将仿真测试分为功能仿真、门级仿真及时序仿真，其中功能仿真的测试对象是采用 RTL 代码描述的模型或者采用 EDA 图形化工具构造的组件；门级仿真的测试对象是经过综合/编译后生成的网表文件；时序仿真的测试对象是布局布线后生成的网表文件和标准延时文件。

按照可编程逻辑器件的开发流程，在测试阶段，应依次实施功能仿真、门级仿真和时序仿真，以保证测试的充分性和完整性。

### 10.3.1 仿真测试的特点

对设计的功能和性能进行验证时，通常有板级验证和仿真测试两种途径可供选择。前者是将设计经综合、布局布线后生成的二进制文件下载至可编程逻辑器件的芯片中，利用信号发生器输入各类信号，然后执行板级测试。该方法的优点在于能够直接验证设计的实际工作情况，接近真实的物理运行环境，可信度最高。而缺点主要有以下方面。

(1) 该方法难以保证验证的充分性，难以充分验证大规模复杂设计的功能和时序正确性。基于硬件环境的验证，难以构造复杂的、异常的、极端的测试场景，难以覆盖所有的功能要求。

(2) 验证环境的搭建相对复杂，验证成本高。真实硬件测试环境的搭建相对软件环境更为复杂，维护、调试的时间开销大，且缺少灵活性，想要尽可能覆盖所有的功能和性能指标，成本较高。

(3) 基于硬件环境的测试相对滞后，通常在设计完成后才能够实施，无法在设计的早期介入。因此，大型复杂设计的验证无法依赖于硬件测试完成。由于在整个 FPGA 开发过程中，硬件测试的时间相对滞后，设计缺陷的修复成本和难度也更大。

(4) 难以发现和定位时序上的错误，由于时序问题具有偶发性和环境敏感性，因而在芯片级的测试中很难发现该类错误。

(5) 难以准确定位缺陷。基于硬件的测试能够观测到的信号均为芯片的外部信号，发现功能不符合要求时，需要查看内部信号的状态才能够定位问题。

仿真测试平台能够不依赖于真实硬件环境即可构造被测设计运行所需要的各种激励，从而全面地验证设计的有效性和正确性。相比之下，仿真测试具有如下优点。

(1) 不需要搭建硬件环境，可以在综合前、综合后、布局布线后等各个设计阶段接入，能够在设计的早期发现设计中潜在的问题，验证成本较低。

(2) 能够方便灵活地提供各种类型的激励信号，复杂和异常测试激励的构造相对容易，便于准确验证被测设计正常或异常的功能处理。

(3) 能够观察到任务模块层次下的各个信号的状态，在出现问题时更容易定位错误。

(4) 便于统计测试的覆盖率情况，能够定量地评估测试的覆盖性，以确保测试的充分性。

(5) 便于大规模复杂设计的验证，仿真测试的灵活性使得构造复杂的信号激励以及复杂测试场景变得更加容易。

(6) 能够发现时序错误。在时序仿真中，可以精确地控制各类信号的时序关系，从而验证外部输入信号时序异常对系统处理的影响，进而发现和定位时序错误。

(7) 适合团队进行协同测试，提高测试的效率。

仿真测试按照层次的不同，可以分成功能仿真(针对设计代码实施)、门级仿真(针对综合后网表实施)和时序仿真(针对布局布线后的网表实施)。

仿真测试主要用于发现以下类别的错误：

(1) 功能实现不满足设计要求；

(2) 时序指标不满足设计要求；

（3）对异常情况的处理措施不完善；
（4）信号读写冲突；
（5）功能缺失或多余。

## 10.3.2　仿真测试平台的组成

针对特定的可编程逻辑器件数字系统或者子模块，需要根据该被测件的接口搭建仿真平台，典型的仿真测试平台的组成如图 10-13 所示。仿真测试平台通常包括时钟模块、激励模块、存储模块、桩模块、信息显示模块、自校验模块等。各个模块的作用如下。

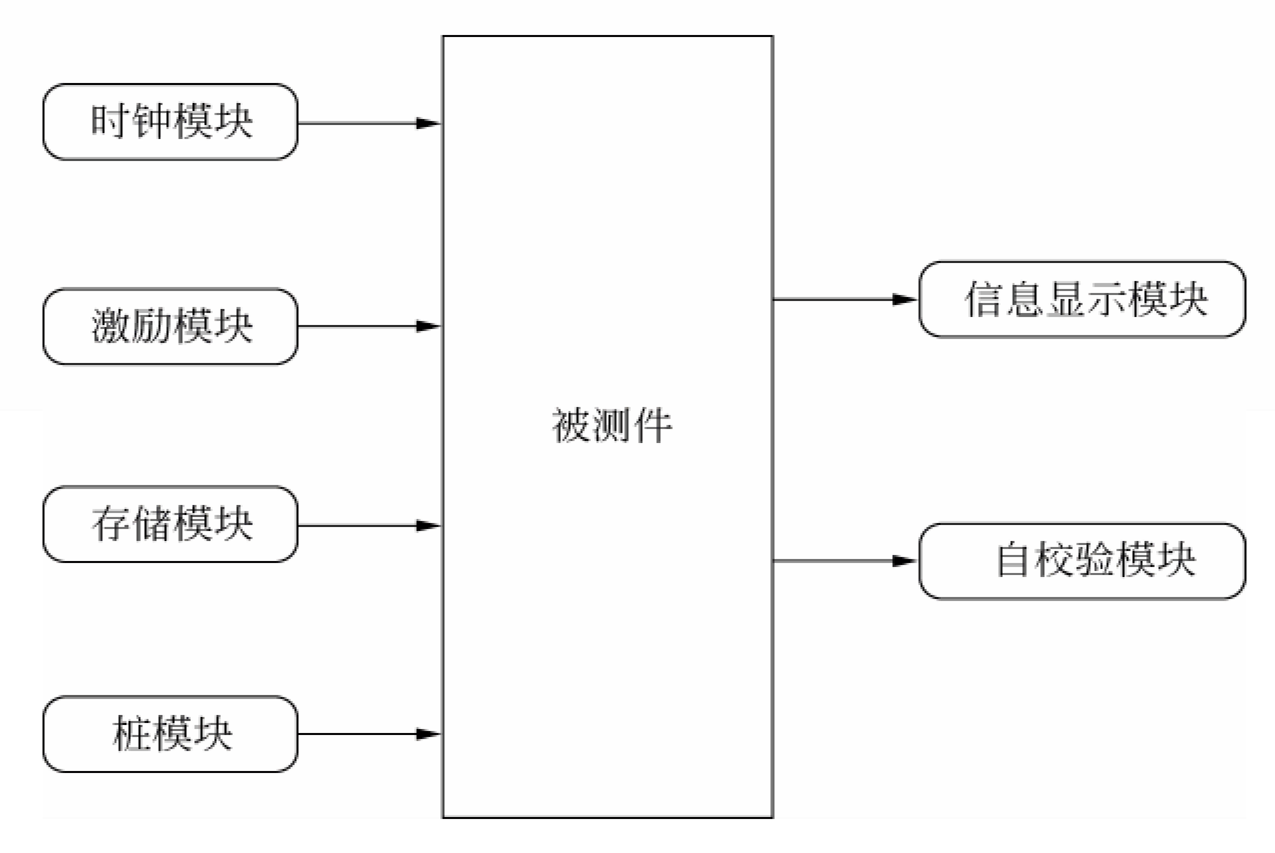



**图 10-13　仿真测试平台的构成**

（1）时钟模块：用于为被测件提供驱动时钟，测试输入的时钟必须与被测试设计所要求的时钟一致，包括时钟频率、占空比和相位关系。

（2）激励模块：用于提供被测件运行所需要的各种激励信号，激励信号是依时间序列添加的。根据时间推进机制的不同可分成绝对时间激励和相对时间激励，前者激励值的变化是参照仿真零时刻的，后者先指定一个初始值，然后在仿真激励触发以前一直等待一个事件。

（3）存储模块：满足被测件进行存储操作的运行要求，通过模拟 SRAM、FLASH 或 ROM 实现。

（4）桩模块：构建被测件所需要调用的其他支持模块或者系统。

（5）信息显示模块：用于输出被测件的运行信息，通过观察波形或者采用 $display 或 $monitor 调用实现。

（6）自校验模块：利用断言或者信号比较，检查被测件的输出与期望结果是否一致。

## 10.3.3　仿真测试的流程

执行 FPGA 的仿真测试时，工作流程如图 10-14 所示（虚线框中给出了仿真平台的编写实例）。图 10-14 的虚线部分给出了对采用 Verilog 语言实现的寄存器移位操作模块 shift_

reg的功能仿真测试流程。

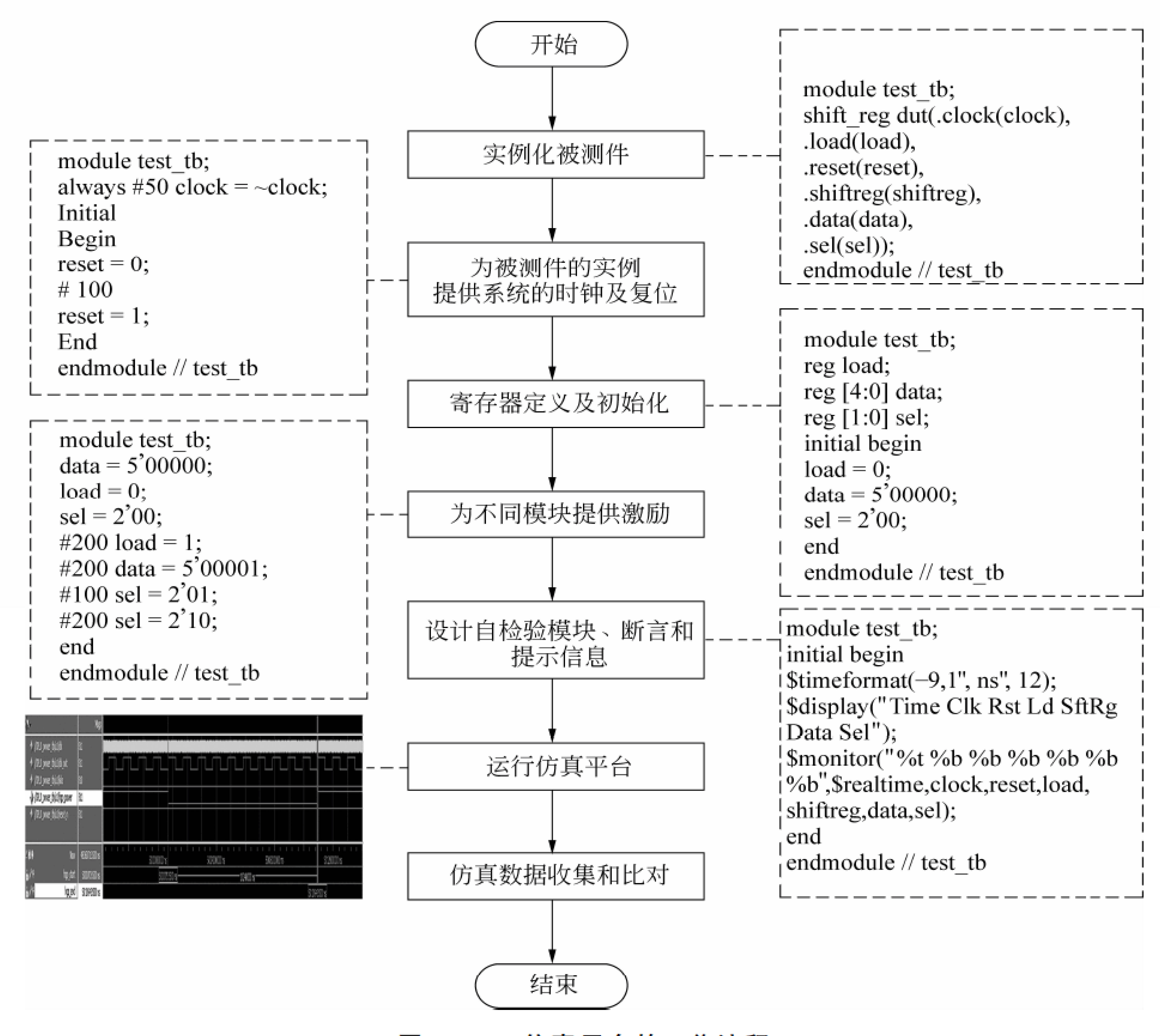



**图 10-14 仿真平台的工作流程**

仿真结果数据收集和比对可以采用波形比对、数据库记录比较或者自定义比对模块输出等方式实现。

在构建仿真测试平台过程中应注意以下9个方面：

(1) 使用统一的编码风格，包括文件头、修订列表、注释、命名规则、文件和目录结构等，以便于测试平台的管理；

(2) 根据测试目标，明确安排激励的发生和变化顺序，确保测试结果的一致性和可重复性；

(3) 尽量利用控制逻辑($ end、$ stop)控制仿真的开始和结束条件，除时钟外，避免使用无限循环；

(4) 将激励分成独立的设计模块，根据模块的相关性进行激励模块的划分，便于激励的控制与测试平台的维护和更新；

(5) 避免显示不重要的数据，以提高仿真的速度，便于数据分析；

(6) 优化仿真中双向信号的使用，例如总线数据，避免因测试平台的错误干扰仿真结果的分析；

（7）仿真中应对寄存器或存储器进行显式的初始化，避免仿真中出现不定态，难以获得正确的结果数据；

（8）采用模块化和参数化，提高测试平台的可扩展性和可维护性；

（9）合理划分测试范畴，确定明确、单一的测试目标，避免在同一个测试平台中观察多种仿真结果，提高仿真的针对性，降低分析难度。

## 10.3.4 功能仿真测试

功能仿真测试是在不包含信号传输延时信息的条件下，利用仿真方法验证设计的逻辑功能是否正确的过程。功能仿真一般应包含如下内容。

（1）依据测试用例的要求，建立功能仿真环境，编制仿真测试激励向量，应满足被测软件外部输入的功能、性能、时序、接口等要求。

（2）在仿真工具中开展功能仿真工作，人工或自动检测仿真结果，并依据判定准则确定测试用例是否通过。

（3）统计语句、条件、分支、表达式等覆盖率信息，对未覆盖的情况进行影响域分析。

在整个的测试流程中，功能仿真的工作量最大，需要编写大量的测试激励，才能保证测试的充分性。为了对测试的充分性加以定量的衡量，引入测试覆盖率的概念，常用的覆盖率指标主要包括语句覆盖率、分支覆盖率、条件覆盖率、表达式覆盖率、翻转覆盖率、状态机状态覆盖率、状态机转移覆盖率等，其中

语句覆盖率＝（至少被执行一次的语句数量）/（可执行的语句总数）

分支覆盖率＝（判定结果至少被评价一次的数量）/（判定结果的总数）

条件覆盖率＝（条件操作数值至少被评价一次的数量）/（条件操作数值的总数）

表达式覆盖率＝（表达式条件操作数值至少被评价一次的数量）/（表达式条件操作数值的总数）

翻转覆盖率＝（被测逻辑节点发生状态翻转一次的状态转换）/（被测逻辑节点所有的状态转换的种类）

状态机状态覆盖率＝（状态机在仿真过程中至少到达一次的状态数量）/（状态机包含的状态总数）

状态机转移覆盖率＝（状态机在仿真过程至少覆盖一次的状态转移路径）/（状态机中包含的所有状态转移路径总数）

在测试过程中，应根据被测对象的关键等级、规模、测试资源和领域要求合理地确定各项覆盖率指标。一般地，对于高关键等级的设计，测试中要求语句覆盖率、分支覆盖率和条件覆盖率均达到 100%。某些模块的覆盖率未达到要求，应说明原因，并分析其影响。对于低关键等级的设计，测试中需要根据需要，选取关键模块，提出较高的测试覆盖率要求。

测试过程中，需要针对测试工具给出的覆盖率信息，不断完善测试平台，添加或修改测试用例，在迭代中提高测试的覆盖率，以提高测试的充分性。

功能仿真测试便于发现被测设计实现中与需求不一致的地方。例如对某个可编程串行通信接口进行功能仿真验证时发现，该接口对地址为 0x3FD 的配置寄存器进行写入时，写入后的寄存器值与 DATA 总线上的预置值不一致，导致寄存器配置错误。具体地，在地址

0x03FDH 向配置寄存器写入值 0x33(DATA 总线上的值),配置寄存器的值由原始值 0x61 变为 0x73,而不是要求的 0x33。功能仿真输出如图 10-15 所示。

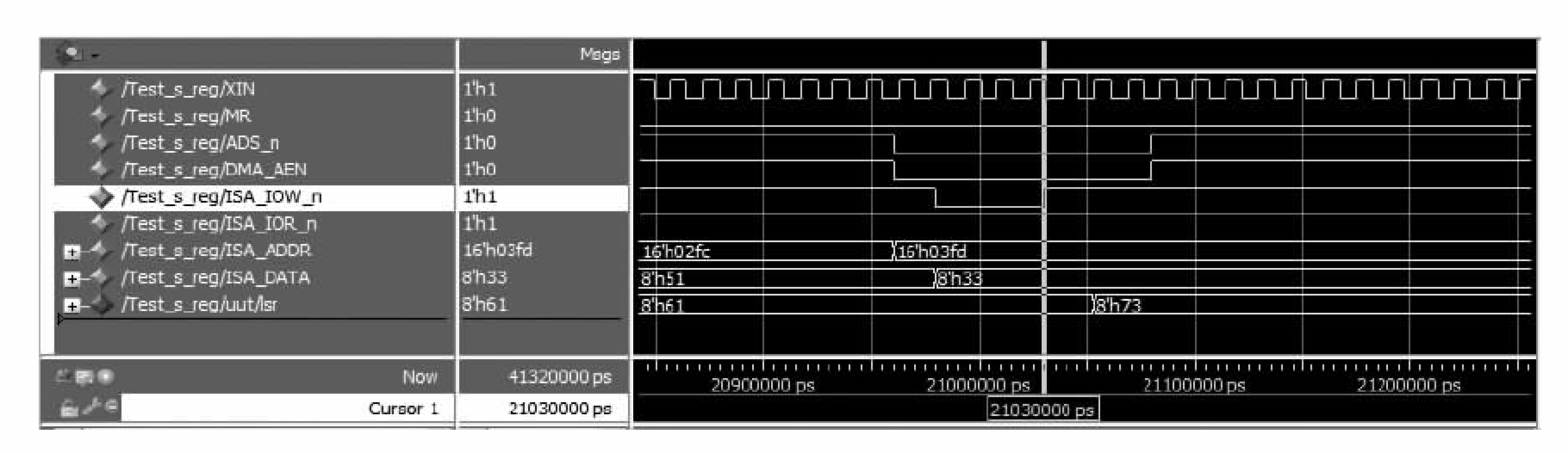



图 10-15　功能仿真实例

## 10.3.5　门级仿真测试

门级仿真是针对逻辑综合后网表文件开展的仿真测试,门级仿真一般包括以下工作内容:

(1) 依据测试用例的要求,建立门级仿真环境,编制仿真激励测试向量,针对逻辑综合后的网表文件开展门级仿真;

(2) 在仿真工具中开展门级仿真工作,人工或自动检测仿真结果。

门级仿真多用于对状态机的测试,主要包括如下内容:

(1) 状态机是否存在多余状态;

(2) 状态机的编码类型是否合理;

(3) 状态机的恢复性是否满足要求;

(4) 状态机的安全性是否满足要求;

(5) 是否存在死状态;

(6) 状态机的防危措施是否得当。

在实际测试中,应重点对包含复杂状态机的设计内容进行门级仿真测试。例如,对某型 1553B 接口控制芯片进行门级仿真测试时发现,在 QuestaSim 中,将 State 寄存器的值强制为一个不存在的值,经过一个运行周期后,将该强制值取消,该设计模块的状态机没有从异常状态中恢复,依然处于“跑飞”状态,无法正常完成数据接收。经排查发现,开发人员设计该状态机时,没有采用安全的状态编码(如格雷码等),状态机进入异常状态时无法重新恢复到正常的状态机模式,进而导致设计失效。

## 10.3.6　时序仿真测试

时序仿真测试是指在包含信号传输的门级延时和布线延时信息的条件下,用仿真方法验证布局布线后功能和时序是否正确的测试。时序测试主要关注以下内容:

(1) 接口时序设计是否合理,满足要求;

(2) 关键信号的输出时序是否正确;

(3) 异步信号的处理时序是否正确;

(4) 不同工况下的信号处理是否正确。

时序测试需要依据 FPGA 片级的输入和输出接口时序要求,构造时序仿真测试用例。重点考察在时序仿真下,当前设计能否正确响应符合时序接口要求的输入信号;设计的输出信号能否满足外部的时序要求;输出信号是否稳定,有无不定态抖动。

实际测试中,时序仿真往往运行与功能仿真相同的测试激励,将被测对象由 RTL 设计模块变为布局布线后的网表文件即可。时序仿真的输出结果与功能仿真不一致或者存在外部接口输出的不定态,则意味着存在时序错误。

图 10-16 为对某数字控制单元执行功能仿真时的输出,各个控制信号(col_en、p_en、r_en、tic_en)的状态是确定的,图 10-17 为对其在典型工况下进行时序仿真时的输出结果,显然各个控制信号的输出出现了不定态,说明当前设计中存在设计错误或时序违规。

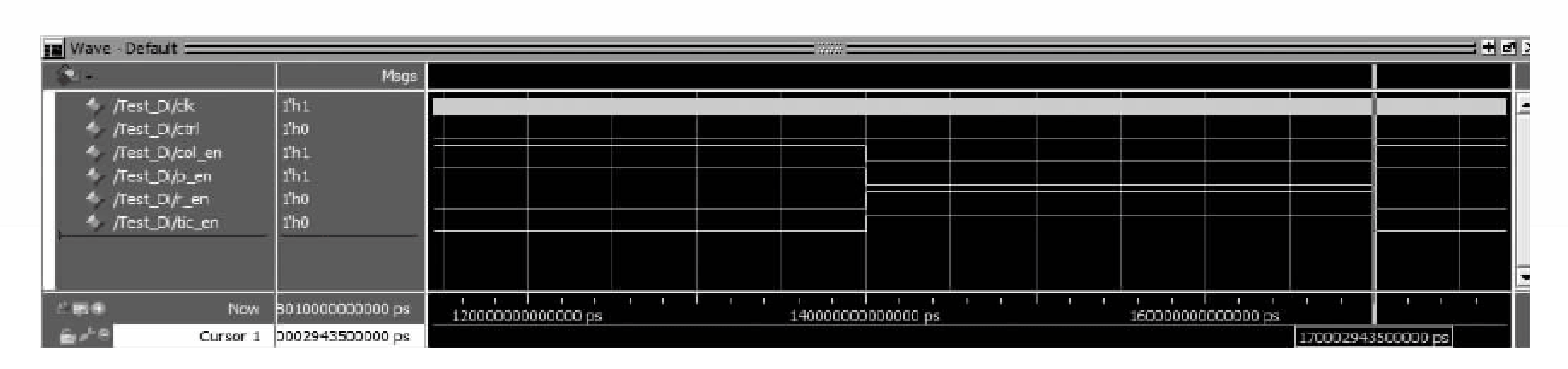


图 10-16 控制信号输出的功能仿真结果

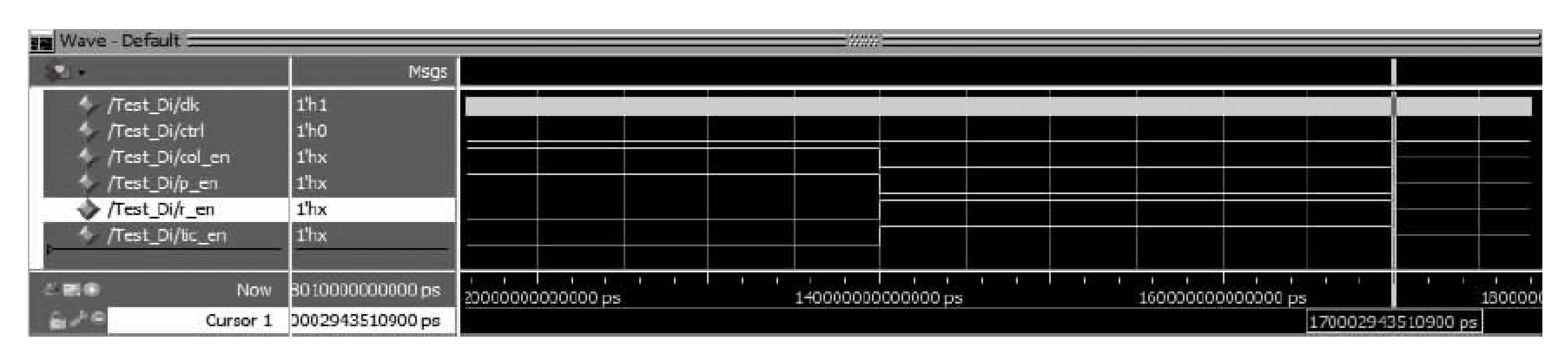


图 10-17 控制信号输出的时序仿真结果

## 10.3.7 仿真测试支持工具

目前,开发人员通常利用芯片厂商提供的集成开发环境(integrate development environment,IDE)完成大部分的仿真验证工作。但是 IDE 开发环境对大规模 FPGA 设计的测试支持不足,存在仿真效率不高、不支持高级验证方法学、不支持跨语言混合设计测试、对测试覆盖率统计支持有限等缺点,无法满足第三方评测的技术要求。为了满足测试的需要,尤其是保证高关键等级软件的测试充分性的要求,需要建立通用型的,能够兼容不同类型芯片的仿真支持平台,应满足如下条件:

(1) 支持主流的芯片厂商 Altera、Xilinx、Actel 等;

(2) 支持大规模 FPGA 设计的测试仿真;

(3) 具有较高的测试效率,仿真时间在可容忍的范围内;

(4) 能够统计测试的功能覆盖、条件覆盖、分支覆盖和语句覆盖情况;

(5) 支持高级验证方法学;

（6）支持混合语言实现的 FPGA 设计的测试；

（7）能够辅助搭建功能仿真和时序仿真的框架；

（8）方便进行 Test bench 的组织和管理。

常用的仿真支持工具主要有 ISE、Quartus Ⅱ、Modelsim、Questasim、VCS 和 IES 等。这些工具是由芯片商为 FPGA 开发者提供的，仅能支持特定芯片平台的 FPGA 设计，无法提供足够的方法学和覆盖率测试支持。而 Questasim 作为 Modelsim 的升级版本，不仅能够方便地针对不同平台的 FPGA 设计进行仿真测试，还能够提供高级验证方法学、混合语言、断言等先进的测试方法。相比其他的仿真测试支持工具，Questasim 具有如下技术优势。

（1）Questasim 能够支持复杂激励信号组合的自动生成，激励情形以约束的方式描述出来。这些约束的随机特征能够显著提高测试平台的设计复用，极大地缩减测试开发和修改的次数，提高了测试设计效率。

（2）Questasim 可以管理并收集代码覆盖率、功能覆盖率和断言覆盖率统计数据，并统一存储在数据库中，通过关联测试覆盖率和测试对象，促进验证进度跟踪和测试资源的合理高效配置。

（3）Questasim 具有测试分析能力，将原始覆盖率数据转变为可操作、易识别的信息，能够识别冗余测试，并基于测试项覆盖率判断是否进行优先测试。

（4）Questasim 通过支持所有主流验证语言和多个验证引擎，并最大限度地发挥断言的技术优势，有效提升验证和调试效率，保证设计质量并改善验证结果的可预测性。断言技术的运用改善了测试的可观测性，便于查找设计中错误的源头，提高了测试的效率。

# 10.4 软硬协同验证

## 10.4.1 验证环境构成

软硬协同验证（SW/HW co-verification）源于逻辑仿真技术，通常由一个硬件执行环境和一个软件验证环境组成，通过时间和命令，采用一些机制在两个环境中进行控制和信息交互，硬件仿真通过运行在工作站上的软件程序进行模拟，通过设定的通信接口与软件执行模块进行信息交互。典型的可编程逻辑器件软硬件协同验证环境如图 10-18 所示。

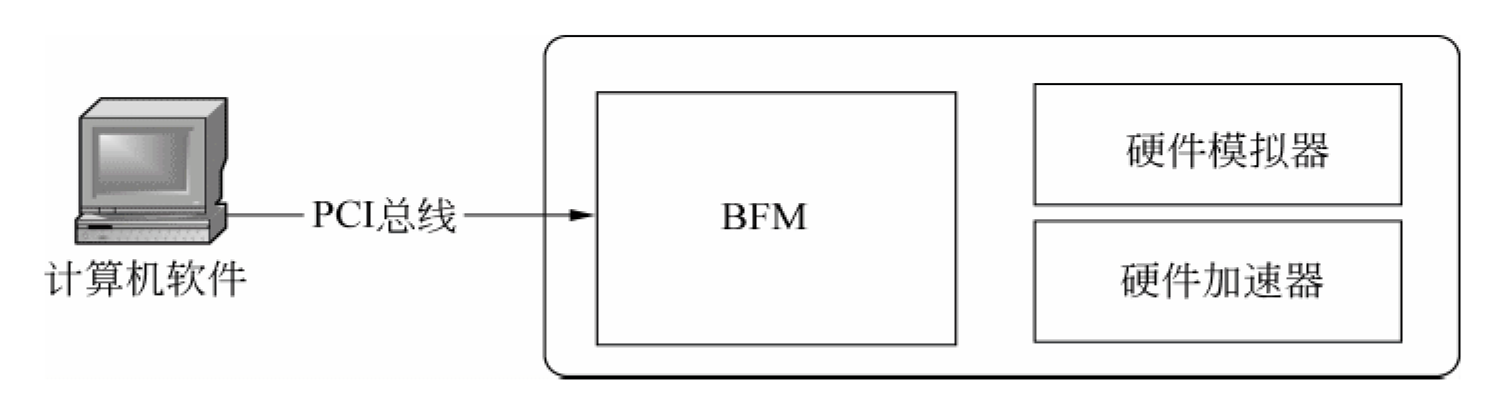



图 10-18　软硬协同验证环境

硬件加速器和硬件模拟器一起构成被测设计（DUT）的运行环境，这种方式能够提供优于纯软件仿真的仿真运行速度以及优于纯硬件验证的信号可观测性，能够综合地验证被测设计的功能和时序。总线功能模型（bus function model，BFM）用于对信号的时序进行抽象和封装，忽略信号的具体时钟周期，将其转化为以事物为单位来进行测试激励的传递，从而

提高测试仿真的运行速度。计算机软件主要完成测试向量的生成，测试数据的注入、事务的调度与控制，计算机通过 PCI 总线与被测设计的加速板连接。

### 10.4.2 支持工具

目前，业界主流的支持软硬协同验证的工具及其特点如表 10-9 所示。

表 10-9 软硬协同支持工具

| 生产厂家 | 产品型号 | 特点 |
|---|---|---|
| MentorGraphics | Seamless | • 对软硬件执行完全的可视化和控制<br>• 在设计早期对硬件/固件/软件的有效调试<br>• 快于逻辑仿真器 1000～10000 倍的动态优化<br>• 支持超过 100 个嵌入式 CPU、DSP 和 FPGA |
| Cadence | Palladium Ⅲ | • 通过仿真加速和仿效，使得验证速度提高 100～100000 倍<br>• 支持软硬件协同验证<br>• 最大容量为 256M 门 ASIC 和 74GB 存储器<br>• 最大验证速度为 2MHz<br>• 最大 I/O 端口数为 61440 个 |
| Synopsys | 扩展型 Confirma | 硬件部分<br>• CHIPit 系列快速原型系统带有一个可实现更大自动化的可编程互联架构，并提供了专为基于事务的验证而优化的类仿真能力<br>• HAPSTM 系列快速原型板为系统验证和嵌入式软件开发提供了较高的性能<br>• 集中了广泛的高速接口和扩展板，确保了各种原型可以被很轻松地定制，并涵盖了一个广泛的应用范围<br>软件工具<br>• CHIPit Manager Pro 原型配置和项目管理软件<br>• 与 SCE-MI(标准协同仿真建模接口)兼容的基于事务的协同验证接口<br>• Certify®多 FPGA 实现和分区软件<br>• Identify® Pro 调试软件，带有 TotalRecalTM 可视度增强技术<br>• Synplify® Premier-技术领先的 FPGA 物理综合工具 |
| Xilinx | ML510 Embedded Development Platform | • XilinxVirtex-5FPGA<br>• 2 * 512MBDDR2<br>• 通过 Rocket IO GTX transceivers 的高速 I/O<br>• JTAG 追踪及调试 |

## 10.5 板级验证

### 10.5.1 作用

板级验证是将整个由可编程逻辑器件组成的数字系统(电路板)作为测试对象，用于验

证系统整体的功能、性能和接口等。一方面，由于系统的真实运行状况不仅与功能逻辑和时序设计有关，还与所选用可编程逻辑器件的芯片类型、时钟指标、温度、电压等物理指标有关，因此，受此影响的一些关键功能、性能指标需要用板级验证的方法进行测试；另一方面，被测系统需要与外部模块、系统进行通信，与之相关的接口兼容性需要利用板级验证的手段加以测试。

## 10.5.2 板级验证的典型环境

进行板级测试时，需要针对被测对象的实际运行环境搭建测试环境，需要利用观测设备进行运行结果的观察。一个典型的板级验证环境如图 10-19 所示。

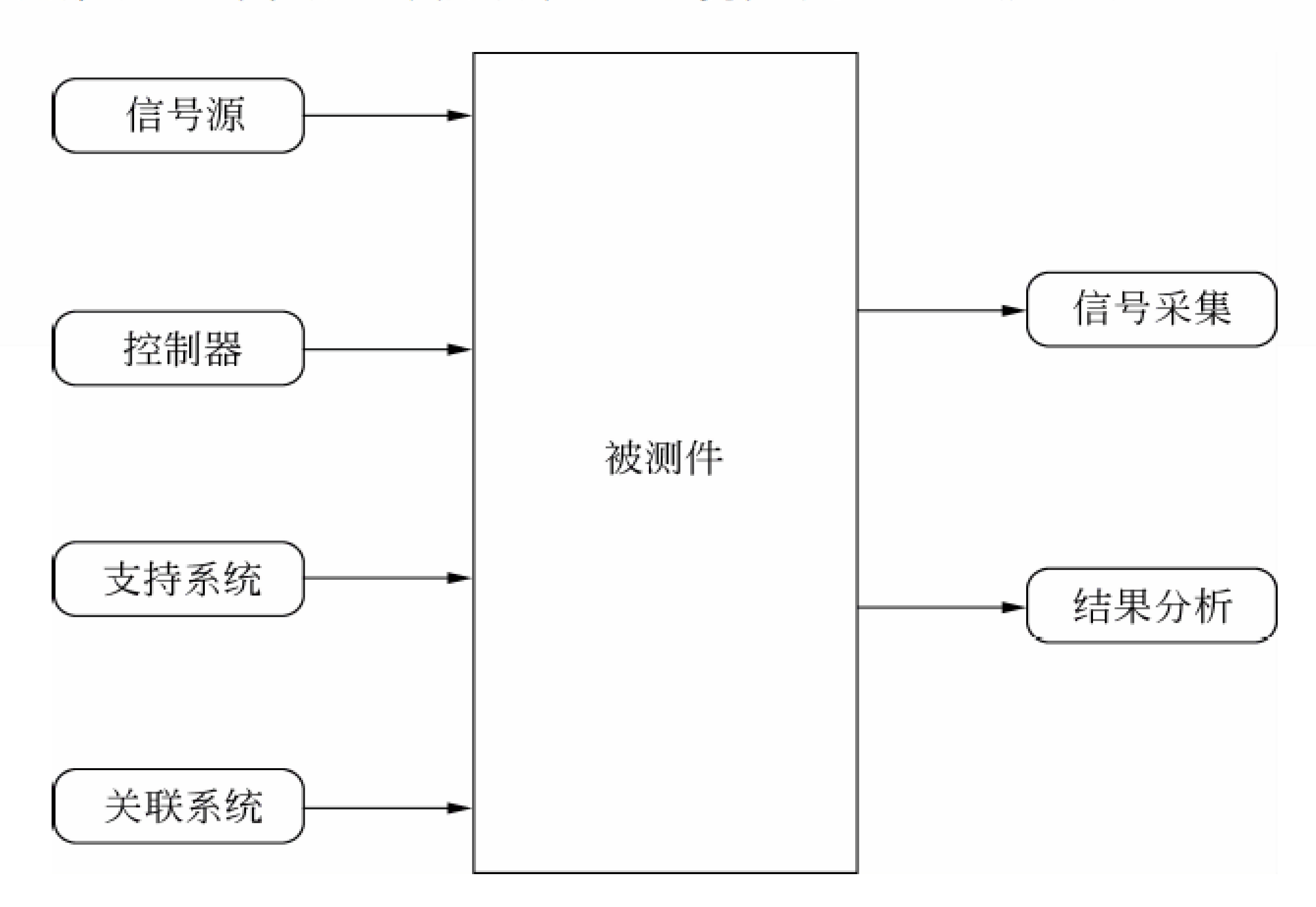



**图 10-19 板级验证环境的组成**

该环境主要由以下几部分组成。

(1) 信号源：用于产生被测件所需要的输入信号，通常利用专用测试仪、回波记录仪、MATLAB 仿真或者微机等产生。

(2) 控制器：主要用于向被测件发送关键的控制信号，如起停信号、复位信号、模式切换信号等。

(3) 支持系统：用于提供被测件运行所需要的温度、电压、电流等支持环境。

(4) 关联系统：指被测系统运行过程中需要进行通信的其他系统或者模块。

(5) 信号采集：通常利用频谱仪、示波器、解码器等仪器观察被测件的输出，以便于进行结果分析。

(6) 结果分析：利用示波器进行波形比对或者利用计算机进行图像、音频或者数据的验证比对，判断结果是否符合预期。

## 10.5.3 板级验证的流程

进行板级验证的流程如图 10-20 所示。

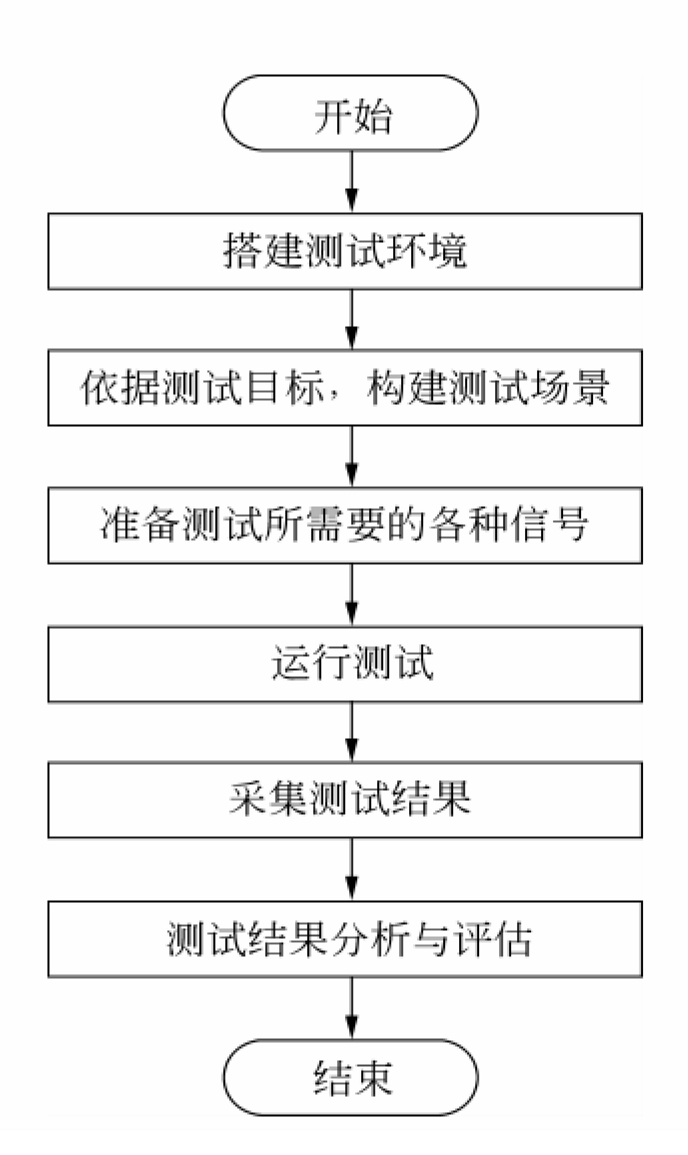



图 10-20　板级验证的流程

# 第 11 章 CHAPTER 11

# 测试工具

## 11.1 概述

软件测试是一项很复杂而费时的工作，仅仅依靠测试人员手工完成是很困难的。所以，必须研究测试工具以帮助测试人员自动或半自动地完成测试。好的测试工具能够提高测试效率从而降低测试成本，选择更高的测试充分性标准进行测试从而提高软件质量。目前市场上的软件测试工具，根据测试工具原理的不同分为静态测试工具、动态测试工具和测试管理工具等。

**1. 静态测试工具**

静态测试工具直接对代码进行分析，不需要运行代码，也不需要对代码编译链接，生成可执行文件。静态测试工具一般是对代码进行语法扫描，找出不符合编码规范的地方，根据某种质量模型评价代码的质量，生成系统的调用关系图等。常用的静态测试工具有Logiscope、PRQA、SpyGlass、PrimeTime、Formalpro、QuestaFormal 等。

**2. 动态测试工具**

动态测试工具与静态测试工具不同，动态测试工具一般采用“插桩”的方式，向代码生成的可执行文件中插入一些监测代码，用来统计程序运行时的数据。其与静态测试工具最大的不同就是动态测试工具要求被测系统实际运行。常用的动态测试工具有 QACenter、WinRunner、JUnit、Testbed、CodeTest、QuestaSim 等。

**3. 测试管理工具**

软件评测是发现软件问题、确保软件质量的有效手段，而软件评测的质量取决于测试技术水平和测试过程管理水平。为保证软件评测项目按时、保质完成，加强评测工作的组织和科学管理显得尤为重要。评测过程管理中存在管理工作节点众多、相互间依赖性强、评测项目信息及数据繁杂、评测文档之间一致性难以保障等多个难题，给评测过程的管理带来了很大的麻烦，且效率低、效果差。因此，对评测过程实施科学、有效地管理，必须借助于得力的自动化工具。一般而言，测试管理工具对测试需求、测试计划、测试用例、测试实施进行管理，并且测试管理工具还包括对缺陷的跟踪管理。常用的测试管理工具有 TestCenter、TP-Manager 等。

除了上述的测试工具外，还有一些专用的测试工具，比如用于数据库测试的 TestBytes、用于性能优化的 EcoScope、用于页面链接测试的 LinkSleuth 等。

# 11.2 静态测试工具

## 11.2.1 Logiscope

### 1. 工具概述

Logiscope 是法国 Telelogic 公司推出的专用于软件质量保证和软件测试的产品。Logiscope 是面向源代码进行工作的，应用于软件的整个生命周期，贯穿于软件开发、代码评审、单元测试、集成测试、系统测试以及软件维护阶段。其主要功能是对软件做质量分析和测试以保证软件的质量，并可实现认证、反向工程和维护，特别适合针对可靠性和安全性要求高的软件项目。

在设计和开发阶段，使用 Logiscope 对软件的体系结构和编码进行确认，可以在尽可能的早期阶段检测那些关键部分，寻找潜在的错误，可帮助项目组成员编制符合企业标准的文档，改进不同开发组之间的交流。在测试阶段使用 Logiscope，可针对软件结构，度量测试覆盖的完整性，评估测试效率，确保满足测试要求，还可以自动生成相应的测试分析报告。在软件维护阶段使用 Logiscope，能够验证软件质量是否已得到保证。对于状态不确定的软件，Logiscope 可以迅速提交软件质量的评估报告，大幅度减少理解性工作，避免非受控修改引发的错误。

该产品的最终目的是评估和提高软件的质量等级，采用基于国际标准度量方法（如 Halstead、McCabe 等）的质量模型对软件进行分析，从软件的编程规则、静态特征和动态测试覆盖等多个方面，量化地定义质量模型，并检查、评估软件质量。

### 2. 产品功能

Logiscope 包括 3 个工具，下面分别介绍其功能。

1）Logiscope RuleChecker

根据工程中定义的编程规则自动检查软件代码错误，Logiscope RuleChecker 可直接定位错误。该工具包含大量标准规则，用户也可定制创建规则，包括结构化编程规则、面向对象编程规则等。具体而言，这些规则有：命名规则，如变量名首字母大写等；控制流规则，如不允许使用 GOTO 语句等。这些规则可以根据实际需要进行选择，也可以按照自己的实际需求更改和添加规则。该工具使用所选规则对源代码逐一进行验证，指出所有不符合编程规则的代码及其违反的规则。

Logiscope 提供编码规则与命名检验，这些规则根据业界标准和经验制订。因此，可建立企业可共同遵循的规则与标准，避免不良的编程习惯及彼此不相容的困扰。同时 Logiscope 还提供规则的裁剪和编辑功能，可以用 Tcl、脚本和编程语言定义新的规则。

2）Logiscope Audit

Logiscope Audit 可以定位错误模块，评估软件质量及复杂程度。该工具提供代码的直观描述，能够自动生成软件文档。该工具以 ISO 9126 模型作为质量评价模型的基础，质量

评价模型描述了从 Halstead、McCabe 的度量方法学和 Verilog 引入的质量方法学中的质量因素(可维护性、可重用性等)和质量准则(可测试性、可读性等),将被评价的软件与规定的质量模型进行比较,并用图形形式显示。因此,质量人员可以把精力集中到需要修改的代码部分,对与质量要素和质量模型不一致的地方给予解释和纠正。具体的图形表示法有以下3种:

(1) 整个应用的体系结构:显示部件之间的关系,评审系统设计;

(2) 具体部件的逻辑结构:通过控制流图显示具体部件的逻辑结构,评审部件的可维护性;

(3) 评价质量模型:通过度量元对整个应用源代码进行度量,并作出 Kiviat 图显示分析结果,对可维护性作出评判。

Logiscope Audit 采用的是包括软件质量标准化组织制定的 ISO 9126 模型在内的质量模型,该模型是一个3层的结构组织,包括:

(1) 质量因素;

(2) 质量准则;

(3) 质量度量元。

质量因素是从用户角度出发,对软件的质量特性进行总体评估;质量准则从软件设计者角度出发,设计为保障质量因素所必须遵循的法则;质量度量元从软件测试者角度出发,验证是否遵循质量准则。一个质量因素由一组质量准则来评估,一个质量准则由一组质量度量元来验证。

Logiscope Audit 将软件与所选的质量模型进行比较,生成软件质量分析报告。内容包括:

(1) 质量报告,Logiscope 根据质量模型,生成相应的软件质量分析报告;

(2) 质量度量元,可清楚分析和观察每个类或方法中的质量度量元的数值,判断其是否合法;

(3) 质量准则,可清楚分析和判断各质量因素所含有的质量准则的数值和合格性;

(4) 质量因素,针对系统层、类层和函数层,分别分析质量因素的合格性和所占百分比;

(5) 程序流程图,控制流图显示算法的逻辑路径。其图形表示适用于评价函数的复杂性;

(6) 程序调用图,调用图显示过程和函数之间的关系,非常适用于检查应用系统的设计;

(7) Kiviat 图,Kiviat 图使质量等级与所选择的参考之间的一致性对比更加可视化。

通过对软件质量进行评估及生成控制流图和调用图,发现最可能发生错误的部分,然后使用度量元及控制流图、调用图等手段做进一步分析。

3) Logiscope TestChecker

Logiscope TestChecker 可以测试覆盖分析,显示没有测试的代码路径,作出基于源码结构分析。该工具直接反馈测试效率和测试进度,协助进行衰退测试;既可在主机上测试,也可在目标板上测试;支持不同的实时操作系统,支持多线程;可累积合并多次测试结果,自

动鉴别低效测试和衰退测试。该工具提供指令覆盖、判定覆盖、MC/DC(条件组合覆盖)和基于应用级的 PPP 覆盖。分析这些覆盖率信息可以提高测试效率,协助进行进一步测试。同时,Logiscope 支持对嵌入式系统的覆盖率分析。

为保证测试的有效性,必须定义准则和策略以判断何时结束测试。准则必须是客观和可量化的元素。根据应用的准则和项目相关的约束,可以定义使用的度量方法和要达到的覆盖率,度量测试的有效性。

TestChecker 产生每个测试的测试覆盖信息和累计信息,并能够根据测试运行情况显示新的测试所反映的测试覆盖情况。被执行过的函数,一旦作了修改需要重新运行时,Logiscope 将会标出。测试时首先要执行的测试是功能性测试,目的是检查所期望的功能是否已实现。在测试初期,测试覆盖比率会迅速增加,一般能达到 70% 的覆盖率。但是,要提高此比率是十分困难的,主要是由于测试覆盖了相同的测试路径。在该阶段需要对测试策略做一些改变,应当执行结构化测试,即要检测没有执行的逻辑路径,定义适当的测试覆盖这些路径。在测试执行期间,当测试策略改变时,可综合运用 TestChecker 和 RuleChecker 帮助用户分析未测试的代码。用户可以显示所关心的代码,并通过对执行未覆盖路径的观察得到有关的信息。信息以图形(控制流图)和文本(伪代码和源文件)的形式提交,并在其间建立导航关联。

Logiscope 支持嵌入式软件测试。众所周知,嵌入式软件测试与非嵌入式软件测试相比要困难一些,因为嵌入式软件开发是用交叉编译方式进行的,在目标机(target)上,不可能有多余的空间记录测试的信息,必须实时地将测试信息通过网线/串口传到宿主机(host)上,并实时在线显示。因此,对源代码的插桩和目标机上的信息收集与回传成为问题的关键。Logiscope 很好地解决了这些技术,成为嵌入式软件测试工具的佼佼者。

#### 3. 产品应用

Logiscope 支持 VxWorks 、pSOS 、VRTX 等实时操作系统。目前,Logiscope 产品在全世界 26 个国家的众多国际知名企业中得到了广泛的应用,其用户涉及通信、电子、航空、国防、汽车、运输、能源及工业过程控制等众多领域。

### 11.2.2　PRQA

#### 1. 工具概述

PRQA 公司成立于 1986 年,总部在英国。PRQA 被世界范围内的高级软件开发人员、行业专家、标准团体认可为编程标准专家,一直致力于通过静态分析来自动化地检查编程标准的遵循并发现软件的缺陷。PRQA 的主要业务是代码完整性管理系统的开发,保证软件质量,提供相关的自动化测试/管理工具,提供专业的咨询和培训业务。其产品以及服务广泛应用于汽车、电子商务、医疗器械、生产和通信等领域。

PRQA 的主要产品包括:QAC/QAC++ ,QA. MISRA C/QA. MISRA C++ 。

QAC/QAC++ 是一个完全自动化的代码静态分析工具,可以提供编码规则检查、代码质量度量、软件结构分析等功能,QAC/QAC++ 以其能全面而准确地发现软件中存在的潜

在问题的能力得到客户的广泛认可。本书重点介绍 QAC。

QAC 是一个基于 C 语言开发环境下用以提高软件产品产量和质量标准的深层次静态分析工具软件。这个软件可以自动识别 C 语言源代码中出现的问题。这些问题主要是由于语言使用过程不安全、过于复杂、无法移植、难以维护或与该行业的代码标准偏离造成的。QAC 能够对许多编译器或其他工具开发软件无法说明的问题提出警告。该工具可以极大地缩减代码检测的时间并能同时加强程序设计人员对 C 语言中不完全为人理解的某些特点的认知。利用 QAC，能够在软件开发的早期阶段发现和解决存在的问题隐患，从而提高代码质量、缩短测试周期，达到双赢的效果。该工具具有以下特点。

(1) 提供深层次的静态分析。工具不但能够迅速有效地检测出语言运用中的错误、已过时用法、程序标准一致性等问题，从而防止在软件开发后期以更昂贵的代价去解决问题，而且还能将工业标准分析度量标准和通俗易懂的报告结合在一起。

(2) 规则可以定制。对于数据库中已有的规则，可以由测试人员决定使用哪项规则或不使用哪项规则，或者是某一个错误等级的规则。对于特殊行业来讲，如果数据库中没有所要求的规则，也可以人工定制，即通过 QAC 提供的定制方法来添加想要的规则。

(3) 可以和开发工具集成。可以和 Visual Studio v6.0、Visual Studio .NET、Tornado 集成，在开发环境中使用 QAC，提高测试效率。

**2. 产品功能**

QAC/QAC++ 的主要功能包括以下 6 个方面。

(1) 代码自动审查。QAC/QAC++ 能够对 C/C++ 代码规则进行自动检查，报告所违反的编程标准和准则，减少代码审查所需的时间，使软件工程师在开发阶段就可以避免代码中的问题，提高代码的质量，缩短后期动态测试的周期。

(2) 科学的质量度量。QAC/QAC++ 提供权威的度量指标分析能力，包括 60 多种 C 语言度量和 20 多种 C++ 度量，为不同成熟度企业的软件质量改进提供客观准确的依据。

(3) 全面的规范支持。QAC/QAC++ 全面支持多种最新编程标准(ISO、MISRA-0：2004、MISRA-C++：2008、JSF、EC++ 等)，及多种其他行业编程规则。QAC/QAC++ 能够发现 1500 多种 C 语言问题、800 多种 C++ 的问题，并提供方便的二次开发接口，可以让软件质量工程师定制符合部门要求的规范。

(4) 灵活的测试管理。QAC/QAC++ 能够对全生命周期的测试过程提供代码质量管理解决方案，通过 QAVerify 插件，开发经理可以随时通过网页监控项目质量趋势、跟踪缺陷状态、定义复合度量、对比分析质量、定制生成质量报告，并和 Baseline 插件一起，进行测试的版本管理和控制。

(5) 强大的结构分析能力。QAC/QAC++ 能够在功能模块、文件引用、函数调用、代码控制流等不同层次，进行软件结构分析和诊断。通过 Structure101 插件，软件设计工程师可以在集成阶段更好地理解软件架构，解析依赖关系、裁汰冗余代码。

(6) 丰富的工具集成。QAC/QAC++ 可以被非常方便地集成到各种 IDE 中，开发工程师在熟悉的环境中就可以进行代码审查，例如 MS VC++、Tornado、Source Insight、GCC、Keil C、C++ Builder、Eclipse、CodeWarrior、Rhapsody 等。QAC 提供了强大的 CLI 接口，可以方便地实现自动化测试脚本。通过所提供的插件包和配置生成器，测试工程师可以快速地搭建起各种嵌入式平台下的测试环境，并能够和业界领先的动态测试工具

VectorCAST 集成在一起，实现强强联合，构成静动态一体化测试环境。

**3. 产品应用**

该产品目前支持的语言有 C、C++、Java & Fortrano；目前可支持的平台有 Microsoft Windows，Sun Solaris，HP-UX，Redhat Linux，Slackware Linux。

对于编译器，QAC 支持几乎所有的主流编译器，QAC 在分析代码时，并不对代码进行编译，只是分析程序运行时要用到的一些编译器信息，如编译器头文件、宏等，以及嵌入式的 CPU 设定等。目前支持的编译器有：Borland C/C++ Builder、Cosmic、Diab、Edison Design Group、GNU C/C++、Green Hills C/C++、IAR、IBM VisualAge C++、Intel、Keil、Metrowerks、Microsoft C/C++、Tasking 等。

该产品具有以下优点：

(1) 缩减软件开发的成本和产品上市的时间；

(2) 降低软件产品质量问题；

(3) 实现代码检测过程自动化，使软件开发和质量检验技术人员提高效率；

(4) 在软件研发的早期阶段识别潜在的软件产品问题和其他可能出错的问题，从而减少产品测试和顾客使用中发现问题的概率；

(5) 具有较好的集成性，能在现有的软件环境下实现安装和卸载；

(6) 自动检测软件产品是否符合某公司或某行业的软件标准和语言安全性方面的要求；

(7) 提高 C 语言代码的编写质量，通过加强软件可靠性、移植性和可维护性 3 方面来减少软件产品未来的维护费用；

(8) 帮助软件开发人员生产高质量的代码；

(9) 支持软件认证，软件研发过程认证和各种质量认证，如 CMM 认证、ISO 9003/EN29003、TickIT、IEC 61508、Def Stan 00-55、DO-178B 等；

(10) 设立了软件质量度量标准，后期代码修改可以得到衡量和比较；

(11) 为软件开发的成本和产量提供依据；

(12) 帮助软件开发相关人员在利用 C 语言编程过程中避免问题。

## 11.2.3 SpyGlass

**1. 概述**

SpyGlass 是由 Atrenta 公司开发的 EDA 软件，可以用来对可编程逻辑器件软件进行编码规则检查和跨时钟域分析。SpyGlass 能够覆盖主流的编码规则集，是目前 FPGA、ASIC 开发领域中编码规则覆盖最广泛的工具之一。

**2. 功能**

1) 编码规则检查

进行编码规则检查时，SpyGlass 软件通过分析 RTL 代码，检查代码的编写模式、规范、风格，将扫描结果与相应的规则库进行匹配，如出现违反相应规则的情况，则根据严重等级予以标识。主要功能如下：

(1) 支持 SPARC、Lint、More Lint、SpyGlass、Sec、OpenMore 等主流规则集;
(2) 覆盖 Do-254 规范中编码相关规则的要求;
(3) 用户可以根据需要选择所要使用的规则集,并可以从各个规则集中抽取特定的规则形成自定义的规则集;
(4) 根据违反规则的类型和程度确定问题等级,分别为 Error、Warning 和 Information 等级,便于用户有侧重点地排查和分析问题;
(5) 针对违规代码模块,给出影响分析和修改建议,便于用户进行问题的定位和分析;
(6) 能够以图形化的方式表示 RTL 代码生成的电路模型;
(7) 统计分析各个设计模块的编码违规情况;
(8) 支持 VHDL、Verilog、System Verilog 等主流设计语言,并支持对混合语言设计的编码规则检查。

2) 跨时钟域分析

进行跨时钟域(CDC)分析时,通过扫描 RTL 设计代码或综合后的设计模型,抽取设计中存在的各种时钟信号,并将其转化为时钟树。然后根据开发人员提供的时钟约束和时钟关联信息,分析各个时钟信号的约束和时钟是否正确,时钟信号的传递和时钟是否违规,存在多个时钟信号的设计模块的时钟信号同步是否正确等。主要功能如下:

(1) 分析和抽取设计中存在的时钟信号,辅助建立完整的时钟约束;
(2) 抽取设计的时钟、复位及 I/O 信号之间的关系,验证测试人员提供的各种约束信息是否完整有效,测试人员可以依据扫描的结果对约束文件进行修改和补充;
(3) 检查设计约束的完整性和正确性;
(4) 对开发人员提供的 RTL 设计进行预综合,根据约束文件对设计的时钟和复位是否存在毛刺、竞争等设计风险进行检查;
(5) 根据约束文件对设计的跨时钟域结构进行抽取并验证所涉及的信号是否进行了正确有效的同步化处理;
(6) 抽取的正确跨时钟域结构,使用自带的仿真引擎进行形式化功能验证,检查各个跨时钟域结构在正确同步的基础上是否满足实际的功能需要;
(7) 以图形化的方式给出存在跨时钟域问题的电路结构、违规模块在整个设计中所处的位置,标注存在跨时钟域问题的各个时钟节点、时钟传递路径等信息,便于设计人员进行问题追踪和排查;
(8) 根据潜在的跨时钟域风险大小,给出问题严重等级,并提供规范的修改建议;
(9) 检查复位信号的正确性,及各个模块对复位信号的相应机制是否合理;
(10) 支持 Verilog、VHDL、System Verilog 等主流设计语言及其混合设计;
(11) 生成跨时钟域分析报告,报告给出存在跨时钟域风险的各个设计模块的具体信息,包含错误等级、风险类型、时钟信息、代码行、原因等信息。

**3. 产品应用**

SpyGlass 能够很方便地执行大规模复杂 FPGA 设计的编码规则检查和跨时钟域分析工作,具有分析效率高、错误信息漏报误报率低的优势。

## 11.2.4 PrimeTime

**1. 概述**

PrimeTime 是 Synopsys 公司推出的一款用于对可编程逻辑器件和 ASIC 电路进行静态时序分析的软件，该软件运行于 Linux/Solaries 环境下，是目前应用比较广泛的静态时序分析工具。

**2. 功能**

PrimeTime 主要完成对以下内容的分析：

(1) 建立和保持时间的检查(setup and hold checks)；

(2) 时钟脉冲宽度的检查；

(3) 时钟门的检查(clock-gating checks)；

(4) 复位信号 recovery 时间和 removal 时间检查；

(5) 无时钟信号输入的时序器件的检查(unclocked registers)；

(6) 未约束的时序端点(unconstrained timing endpoints)；

(7) 多时钟输入的时序器件的检查(multiple clocked registers)；

(8) 组合反馈回路识别与分析(combinational feedback loops)；

(9) 基于设计规则的检查，包括对最大电容、最大传输时间、最大扇出的检查；

(10) 最大延迟路径的时序余量分析；

(11) 时钟偏移的影响分析与评估；

(12) 不同工况下时钟余量分析。

**3. 产品应用**

PrimeTime 采取穷尽的方法提取整个 FPGA 设计的时序电路，通过计算信号在各个路径上的传播延迟，对最大延迟和最短路径进行分析，计算时序余量，检查设计中是否存在违背时序约束的错误。PrimeTime 能够快速高效地对大规模 FPGA 设计进行时序分析，极大地节省时序仿真的开销，提高开发和验证效率。

## 11.2.5 Formalpro

**1. 概述**

Formalpro 是一个形式化验证工具，用于检查两个设计(或对等实体)在逻辑功能上的是否等价，是一款常用的逻辑等效性检查工具。逻辑等效性检查是验证两个设计在逻辑上是否等价的技术。该技术可以验证 RTL、综合后网表、门级网表、布局布线后的网表在逻辑上是否等价。

**2. 功能**

Formalpro 通过完整提取设计中不同层次的参考点，参照名称和接口匹配这些参考点，验证不同层次的设计是否保持逻辑上的一致性。该工具的主要功能如下：

(1) 支持多种主流 RTL 语言，如 VHDL、Verilog、System Verilog 等主流设计语言，作

为验证端输入；

(2) 支持渐进式的设计验证，验证过程中仅重新编译更动后的代码，提高验证效率；

(3) 验证过程中的任何时间点均可暂停和开始；

(4) 辅助验证人员手工编写匹配文件，支持正则表达式匹配方法；

(5) 支持图形界面和命令行界面；

(6) 以图形化的方式完整显示被验证的两个设计的电路模型，高亮显示不匹配的电路结构、RTL 代码等信息；

(7) 明确表示不匹配点，以图形化方式标注不匹配的参考点，便于错误定位和双向追踪；

(8) 支持调试工作模式，便于验证人员更加直接地判定匹配情况，并动态调整匹配文件；

(9) 能够给出完整的匹配报告，详细给出匹配情况和分析结果。

**3. 产品应用**

Formalpro 能够帮助设计者很快地检测出设计中的错误并将之隔离，大大缩短验证所需要的时间。比如，开发人员在对 RTL 代码级别的设计进行了充分功能仿真后，只需使用逻辑等效性检查技术验证综合后的网表文件或布局布线后的网表文件与 RTL 代码在逻辑上是否等价即可，从而保证 FPGA 设计功能验证的充分性，节省了进行门级仿真、时序仿真耗费的大量时间。

### 11.2.6 其他静态测试工具

其他静态测试工具有：

(1) McCabe 公司的 McCabe IQ，支持 C、C++、Java、Ada、Visual Basic 和.NET，用于静态结构分析、代码复杂度和覆盖率分析，包含 McCabe Test、McCabe QA、McCabe Reengineering 等组件；

(2) Gimpel 公司的 PC-LINT，支持 C、C++；

(3) PolySpace 公司的 PolySpace，支持 C、C++、Ada，能够进行代码静态分析。

## 11.3 动态测试工具

### 11.3.1 QACenter

**1. 工具概述**

QACenter 用于帮助测试人员创建一个快速、可重用的测试过程，该工具自动帮助管理测试过程，快速分析和调试程序，针对回归、强度、单元、并发、集成、移植、容量和负载建立测试用例，自动执行测试和产生文档结果。

**2. 产品功能**

QACenter 主要包括以下 5 个模块。

1）功能测试工具 QARun

在 QACenter 测试产品套件中，QARun 主要用于客户/服务器应用客户端的功能测试。在功能测试中主要包括对应用的 GUI（图形用户界面）的测试及客户端事物逻辑的测试。现在 RAD（快速应用开发）方式开发的应用，由于开发的速度比较快，可支持对用户多变的需求不断调整应用，所以要求对软件要有更严格的测试。由于不断变化的需求将导致应用不同版本的产生，每一个版本都需要测试，每一个被调整的内容往往最容易隐含错误，所以回归测试是测试中最重要的阶段，而回归测试通过手工方式是很难达到的，工具在这方面可以大大地提高测试效率，使测试更具完整性。

QARun 的测试实现方式是通过鼠标移动、键盘点击操作被测软件，得到相应的测试脚本，对该脚本可以进行编辑和调试。在记录的过程中可针对被测软件中所包含的功能点进行基线值的建立，换句话说就是在插入检查点的同时建立期望值。在这里检查点是目标系统的一个特殊方面在一特定点的期望状态。通常，检查点在 QARun 提示目标系统执行一系列事件之后被执行。检查点用于确定实际结果与期望结果是否相同。

2）性能测试工具 QALoad

QALoad 是企业范围的负载测试工具，该工具支持的范围广、测试内容多，可以帮助软件测试人员、开发人员和系统管理人员对分布式的应用执行有效的负载测试。负载测试能够模拟大批量用户的活动，从而发现大量用户负载下对 C/S 系统的影响。

3）应用可用性管理工具 EcoTools

在性能测试完成之后，需要对可用性状况如何实施分析。很多因素能够影响应用的可用性。用户的桌面、网络、应用的服务器、数据库环境和它们的各种各样的子组件都链接在一起，任何一个组件都可能引起整个应用对最终用户的不可用。

EcoTools 能够解决应用可用性中计划、管理、监控和报告的问题，该工具能够立即在测试或生产环境中激活、计划和管理以商务为中心的应用的可用性，能够支持一些主流成型的应用，如 SAP、PeopleSoft、Baan、Oracle、UNIFACE、LotusNotes 以及其他定制的应用。

EcoTools 有数百个 Agents 可以监控服务器资源，能够监控 Windows、UNIX、Oracle、Sybase、SQL Server 等。通过使用 QALoad 与 EcoTools，可以在系统生成一个负载，同时监控资源的利用问题。QALoad 是一个极好的性能测试工具，但不承担诊断问题的工作，而 QALoad 与 EcoTools 集成则为所有加载测试提供全面的解决方案。QALoad 与 EcoTools 集成允许在图形中查看 EcoTools 资源利用数据，可以使用 QALoad 的分析组件创建。

4）应用性能优化工具 EcoScope

EcoScope 是一套定位于应用及其所依赖的所有网络计算资源的解决方案。EcoScope 可以提供应用视图，并标出应用是如何与基础架构相关联的。这种视图是其他网络管理工具所不能提供的。EcoScope 能解决在大型企业复杂环境下分析与测量应用性能的难题。通过提供应用的性能级别及其支撑架构的信息，EcoScope 能帮助相关部门就如何提高应用性能提出多方面的决策方案。该工具的主要特点如下。

（1）贯穿整个应用生命周期的性能分析。EcoScope 使用综合软件探测技术无干扰地监控网络，可自动发现应用、跟踪在 LAN/WAN 上的应用流量、采集详细的性能指标。EcoScope 将这些信息关联到一个交互式用户界面（Interactive Viewer）中，自动识别低性能的应用、受影响的服务器与用户、性能低下的程度。Interactive Viewer 允许用户以一种智

能方式访问大量的 EcoScope 数据，能很快地找到性能问题的根源，并在几小时内解决令人烦恼的性能问题。另外，EcoScope 的长期数据采集能使我们通过预先趋势分析和策略规划预测到未来的问题。

(2) 确保成功部署新应用。EcoScope 允许使用从运行网络中采集到的实际数据来创建一个测试环境。利用此环境，可以在不影响其他应用的情况下，测量新应用在已存架构中的适应性(即网络能力)，还可测量出与网络共享资源的可交互性。EcoScope 能揭示性能问题，如低伸缩性或瓶颈，能调整应用和定位基础架构上的缺陷。一旦性能得到了提高，EcoScope 可以重新评估，验证应用是否达到了预期目的。这些指标数据可用来作为部署应用的基准，以确保达到预期目标。

(3) 维护性能的服务水平。EcoScope 性能评分卡能很容易地显示出关键应用每时每刻是如何运行的，以及它们是否达到了预期的服务水平。对于必须满足服务水平协议(SLAs)的应用，EcoScope 能为之设置性能要求，并监控是否有偏离。如果一个应用超出了性能的上下限，EcoScope 将认为服务水平异常，并根据受影响用户的数量和性能降低的时间长短细分问题的严重程度。这些信息使 IT 维护人员能优先关注对业务影响最大的应用问题。EcoScope 性能评分卡以图形方式按时间周期显示响应时间和流量，以及受应用影响的关键服务器和最终用户。在评分卡中，能通过比较和关联这些信息，确定应用使用量、响应时间、特定的最终用户和服务器之间的因果关系。在业务被阻碍前，跟踪每天的变化趋势，控制性能波动，快速找出性能瓶颈。EcoScope 一旦发现性能低下的应用，将提供详细信息来隔离造成瓶颈的来源。EcoScope 图形化界面使用户交互地观察单个受影响的工作站、服务器及网段。EcoScope 提供的大量信息有助于进行问题根源的分析，确定问题扩散的原因、受影响的服务器和用户及其性能受损是否有共性。EcoScope 对瓶颈的分析不限于网络基础架构和资源，还包括其他关键计算资源，如桌面和服务器。

(4) 加速问题检测与纠正的高级功能。完善的 EcoScope 技术被动地监视网络，能收集到关于应用与协议的独特信息，不只包括 IP 与 IPX 流量，可以更好地分析与排除应用的性能问题。EcoScope 可自动发现几百种打包的内部应用，如 SAP/R3、MS Exchange、Oracle、SNA LU2 与 LU6.2、Web、IPX/SPX 和 UNIX NOS。不像其他产品需要预先配置才能识别应用，EcoScope 跟踪 LAN/WAN 架构中的应用流量，并显示出应用使用的流量最大的路径及某个服务器的特定路径。EcoScope 通过收集 3 类指标数据提供应用性能的完全视图：会话层响应时间、业务交易响应时间和应用流量。EcoScope 的内置智能技术可识别组成业务交易的 Oracle 与 SQLServer 谓词的不同独特标志，并跟踪其响应时间。

(5) 定制视图有助于高效地分析数据。EcoScope 将信息关联起来并显示到一个单一的交互式用户界面上。这个界面允许按应用或用户来灵活地创建定制的逻辑数据视图，能以最有效的方式来分析信息。这就可以用多种视图显示来自于跨越地理和部门界限的大企业的数据。EcoScope 能把历史信息导出到建模和仿真工具，如 CACI、NetMaker。这些工具可描绘发展趋势和模拟未来的增长。这将使用户能明白未来的瓶颈在哪里，更重要的是，什么时候它将威胁应用的服务水平。

5) TESTBytes

在数据库开发的过程中，为了测试应用程序对数据库的访问，应当在数据库中生成测试

用数据，我们可能会发现当数据库中只有少量的数据时程序可能没有问题，但是当真正投入到运用中产生了大量数据就出现问题了，这往往是由于程序的编写没有达到一些功能，所以一定及早地通过在数据库中生成大量数据来帮助开发人员尽快完善这部分功能和性能。但是如何生成大量测试数据呢？长期以来这些工作是靠手工来完成的，要占用有经验的开发和测试人员大量宝贵时间。TESTBytes 是一个用于自动生成测试数据的强大易用的工具，通过简单的单击式操作，就可以确定需要生成的数据类型(包括特殊字符的定制)，并通过与数据库的连接来自动生成数百万行的正确的测试数据，可以极大地提高数据库开发人员、测试人员、数据仓库开发人员、应用开发人员的工作效率。

**3. 产品应用**

QACenter 应用十分广泛，支持 DB2，DCOM，ODBC，ORACLE，NETLoad，Corba，QARun，SAP，SQLServer，Sybase，Telnet，TUXEDO，UNIFACE，WinSock，WWW 等。

该产品操作简便。以 QALoad 为例，测试人员只需操作被测应用，执行性能关键的事务处理，然后在 QALoad 脚本中通过服务器上应用调用的需求类型开发这些事务处理。QALoad 的测试脚本开发由捕获会话、转换捕获会话到脚本以及修改和编译脚本等一系列的过程组成。一旦脚本编译通过后，使用 QALoad 把脚本分配至测试环境中相应的机器上，可模拟大量用户的并发操作，实施应用负载测试，从而减少了以往大量的人工工作，节省了时间，提高了效率。

## 11.3.2　WinRunner

**1. 工具概述**

Mercury Interactive 公司的 WinRunner 工具是一种企业级的功能测试工具，用于检测应用程序是否能够达到预期的功能及正常运行。由于 C/S 结构的软件功能增加越来越快，QA 部门测试难度越来越大，手工测试已经跟不上这种发展趋势。WinRunner 可以帮助测试人员自动处理从测试开发到测试执行的整个过程。测试人员可以创建可修改和可复用的测试脚本，而不用担心软件功能模块的变更，只需要让计算机自动执行这些脚本，就能轻而易举地发现软件中的错误，从而确保软件的质量。

在软件操作中单击 GUI(图形用户界面)对象时，WinRunner 会用一种类 C 的测试脚本语言(TSL)生成一个测试脚本，测试人员可以用手工编程的方法编辑这个脚本。WinRunner 的功能生成器(function generator)可以帮助测试人员快速简便地在已录制的测试中添加功能。WinRunner 包括两种录制测试的模式。

(1) 环境判断模式。这种模式根据测试人员选取的 GUI 对象(如窗体、清单、按钮等)把对软件的操作动作录制下来，并忽略这些对象在屏幕上的物理位置。每一次对被测软件进行操作，测试脚本中的脚本语言会描述测试人员选取的对象和操作动作。录制时间时，WinRunner 会对测试人员选取的每个对象做唯一描述并写入 GUI map(映射)中。GUI map 和测试脚本被分开保存维护。软件用户界面发生变化时，测试人员只需更新 GUI map。这样一来，环境感应模式的测试脚本将非常容易地被重复使用。执行测试只需要回放测试脚本。WinRunner 能够模拟一个用户使用鼠标选取对象、用键盘输入数据，它从 GUI map 中读取对象描述，并在被测软件中查找符合这些描述的对象，可以在同一个窗体

中找到这些对象，即使它们的位置发生过变化。

(2) 模拟模式。这种模式能够记录鼠标点击、键盘输入和鼠标在二维平面上(x 轴和 y 轴)的精确运动轨迹。执行测试时，WinRunner 让鼠标根据轨迹运动。这种模式对于那些需要追踪鼠标运动的测试非常有用，例如画图软件。

**2. 产品功能**

WinRunner 将测试过程分 6 个步骤。

(1) 创建 GUI map。使用 RapidTest Script Wizard(快速测试脚本向导)回顾软件用户界面，并系统地把每个 GUI 对象的描述添加到 GUI map 中。也可以在录制测试的时候，通过单击对象把对单个对象的描述添加到 GUI map 中。

(2) 创建测试。通过录制、编程的方式创建测试脚本。录制测试时，在需要检查软件反应的地方插入检查点来检查 GUI 对象、位图(bitmap)和数据库。在这个过程中，WinRunner 捕捉数据，并作为期望结果储存下来。

(3) 调试测试。测试人员可以先在调试模式下运行脚本，也可以设置中断点，监测变量，控制 WinRunner 识别和隔离错误。调试结果能够保存下来，一旦调试结束就可以删除。

(4) 执行测试。测试人员在检验模式下测试被测软件。WinRunner 在脚本运行中遇到检查点后，就把当前数据和前期捕捉的期望值进行比较，发现不符合的情况就记录下来。

(5) 查看测试结果。每次测试结束，WinRunner 会把结果显示在报告中。报告会详述测试执行过程中发生的所有主要事件，如检查点、错误信息、系统信息或用户信息。如果在检查点发现问题，测试人员可以在测试结果窗口查看预期结果和实测结果。如果是位图不符合，也可以查看用于显示预期值和实测结果之间差异的位图。

(6) 报告发现的错误。如果由于测试中发现错误而造成测试运行失败，测试人员可以直接从测试结果窗口报告错误信息。

**3. 产品应用**

企业级应用(包括 Web 应用系统、ERP 系统、CRM 系统等)在发布之前，升级之后的版本都要经过测试，确保所有功能都能正常运行。如何有效地测试不断升级更新且不同环境的应用系统，是每个公司都会面临的问题。人工测试的工作量大，还要额外的时间来培训新的测试人员，如果时间或资源有限，这个问题会更加棘手。为了确保那些复杂的企业级应用在不同环境下都能正常可靠地运行，需要一个能简单操作的测试工具来自动完成应用程序的功能性测试。通过自动录制、检测和回放用户的应用操作，WinRunner 能够有效地帮助测试人员对复杂的企业级应用的不同版本进行测试，提高测试人员的工作效率和质量，确保跨平台的、复杂的企业级应用无故障发布及长期稳定运行。

## 11.3.3 JUnit

**1. 工具概述**

目前的最流行的单元测试工具是 xUnit 系列框架，根据语言不同分为 JUnit(java)、CppUnit(C++)、DUnit (Delphi)、NUnit(.net)、PhpUnit(Php)等。该测试框架的第一个和最杰出的应用就是由 Erich Gamma 和 Kent Beck 提供的开放源代码的 JUnit。JUnit 是一

个开发源代码的 Java 测试框架，用于编写和运行可重复的测试。它是用于单元测试框架体系 xUnit 的一个实例（用于 Java 语言），主要用于白盒测试、回归测试。

**2. 产品功能**

JUnit 是为 Java 程序开发者实现单元测试的一个框架，能使 Java 单元测试更规范有效，并且更有利于测试的集成。

1）TestCase 测试类

JUnit 框架中通过 TestCase 实现单元测试，TestCase 继承了 Assert 类，也就是说在 TestCase 类以及子类中可以直接使用 JUnit 框架所提供的断言。另外，TestCase 实现了 Test 接口，Test 接口的 run 方法将会运行一个测试，并返回结果。

如果需要使用 JUnit 做单元测试，可以继承 TestCase，每一个 TestCase 的子类可以作为独立的测试用例，从而实现测试的目的。

2）TestSuite 测试组

一个 TestCase 用例会包含多个 test 开头的测试函数，而一个应用中，会包括若干个 TestCase 的用例，通常用户会运行应用中的所有的测试，从而确保应用是可以运行的。有些测试是非常耗费时间的，如果每次进行一个小的改动就运行所有的测试，这是很多开发人员不可接受的，这时就需要组合一些 TestCase 用例以及 TestCase 中的方法，把测试分成很多的组，每次只针对特定的组进行测试。

JUnit 框架提供了 TestSuite 套件来组合测试类及测试方法，TestSuite 实现了 addTestSuite 方法和 addTest 方法，addTestSuite 和 addTest 能添加一个测试类或另一个 TestSuite 到当前组中，通过这种组合，用户能够把要测试的用例及方法分成一个组，最后组成一个测试的套件。

3）SetUp 与 TearDown

通常每个测试的运行都应该是独立的，从而就可以在任何时候，以任意的顺序运行每个单独的测试。然而有些时候需要一些全局的环境变量设置，每个测试用例都可以用到这些设置，而不必每个测试方法运行前都重新设置，这就需要在测试用例运行前使用不同的初始化方式。

测试用例的每一个方法的执行都是相互独立的，每个 test 方法执行之前，JUnit 框架会执行相应的初始化方法，从而保证每个 test 方法的执行和其他 test 方法之间没有关系。当 test 方法执行后，JUnit 框架也会自动执行清理方法。测试方法的初始化和清理工作是通过实现 TestCase 子类的 SetUp 和 TearDown 方法完成的。

要分清楚 TestCase 和 TestSuite 中方法的执行顺序，初始化条件，这对设计测试项目以及程序的设计都是非常重要的。

4）使用 Eclipse 进行单元测试

Eclipse 对 JUnit 提供了完美的支持，开发人员可以通过 Eclipse 自动生成单元测试的框架，并且能够运行测试代码。

5）创建测试用例

在 Eclipse 中，如果要为某一个被测试的类创建一个单元测试类，可以选择新建向导，打开新建文件对话框，在新建文件对话框中选择 JUnit 下面的“JUnit Test Case”选项，打开创建 TestCase 的对话框。

在创建 TestCase 的对话框中，可以选择是否创建 SetUp 和 TearDown 方法等，另外还能够选择为哪一个类创建对应的单元测试。选择 Next 按钮，可以为被测试的类选择方法加入到单元测试中。选择 Finish 按钮，Eclipse 将会自动生成选择方法的相关定义。

**3. 产品应用**

JUnit 的优点主要有：

(1) 可以使测试代码与产品代码分开；

(2) 针对某一个类的测试代码通过较少的改动便可以应用于另一个类的测试；

(3) 易于集成到测试人员的构建过程中，JUnit 和 Ant 的结合可以实施增量开发；

(4) JUnit 是公开源代码的，可以进行二次开发；

(5) 可以方便地对 JUnit 进行扩展。

随着项目的进展，项目的规模在不断地膨胀，为了保证项目的质量，有计划地执行全面的单元测试是非常有必要的。但单靠 JUnit 提供的测试套件很难胜任这项工作，因为项目中单元测试类的个数在不停地增加，测试套件却无法动态地识别新加入的单元测试类，需要手动修改测试套件，因此与其他构建工具相结合共同创建自动化单元测试方案成为众多项目的选择。

## 11.3.4 Testbed

**1. 工具概述**

Testbed 由 LDRA 公司开发，LDRA 公司是专业性软件测试工具与测试技术、咨询服务提供者，成立于 1975 年，具有丰富的软件测试经验，其总部位于英国利物浦。

**2. 产品功能**

LDRA Testbed/TBrun 软件测试产品功能如下。

1) 静态分析功能

(1) 编程标准验证。编程标准验证是高可靠性软件开发不可缺少的软件质量保证方法，编程规则由软件项目管理者根据自身项目的特点，并参考现有的成熟的软件编程标准制定，如 DERA(欧洲防务标准)、MISRA(汽车软件标准)。LDRA Testbed 依据此规则搜索应用程序，并判断代码是否违反所制定的编程规则。LDRA Testbed 报告所有违反编程规则的代码，并以文本方式或图形反标注的方式显示。测试人员或编程人员可根据显示的信息对违反编程规则的代码进行修改。

(2) 软件度量分析和质量标准验证。对于软件开发工程师、项目负责人及高级管理者来说，软件质量的管理与监控是非常困难且费时的。LDRA Testbed 很好地解决了这一问题，使得管理者很容易收集正在开发的软件系统的相关信息并判断软件是否满足软件质量标准要求，从而达到对软件项目的质量跟踪与控制，用户可基于现行软件标准自行定义适合本系统或项目的软件质量模型。LDRA Testbed 支持下列主要软件度量元分析：控制流节点度量(control flow knots)、LCSAJ 密度度量(LCSAJ density)、扇入/扇出度量、循环深度度量、McCabe 圈复杂度、Halstead 软件科学度量、McCabe Essential 复杂度、注释行度量、

代码可达性度量等。

(3) 静态数据流分析。LDRA Testbed 分析软件中全局变量、局域变量及过程参数的使用状况,并以图形、HTML 或 ASCII 文本报告方式表示,清晰地识别出变量使用引起的软件错误,此种方法既可使用于单元级,亦可使用于集成级、系统级。研究结果表明:数据流分析技术是查找软件错误最有效的途径或方法之一。

(4) 信息流分析。信息流是在数据流分析基础上对数据变量之间的关系作进一步分析,此分析方法已列入高可靠性软件测试标准,为 DEF-STAN 00-55 软件测试标准所采用。

2) 动态分析功能

(1) 源代码自动插桩。LDRA Testbed 可自动对被测软件进行代码插桩,以获得被测软件的动态执行信息,供 Testbed 作动态测试分析。被插桩的代码可为主机平台软件,亦可为嵌入式目标机平台。

(2) 覆盖率分析。如果在进行软件测试时不对代码覆盖率进行监控,有可能在未被执行的代码中遗留软件错误,因此在软件测试过程中有效地监控代码覆盖率是提高软件测试有效性的一项重要途径。LDRA Testbed 可提供如下代码覆盖率指标:语句覆盖(statement)、分支/判定覆盖(branch/decision)、LCSAJ 覆盖(linear code sequence and jump segents)、过程/函数调用覆盖(procedure/function call)、分支条件覆盖(branch condition)、分支条件组合覆盖(branch condition combination)、修正条件/判定覆盖(modified condition/decision)、动态数据流覆盖(dynamic data flow)。通过 LDRA Testbed 对被测软件进行代码覆盖率指标分析,可制定出相应的软件测试策略以达到期望的代码覆盖率要求。这将大大提高对被测软件(或代码)的信心。

(3) 断言分析。LDRA Testbed 提供断言分析功能,使用此功能测试人员可确认被测软件在动态测试过程中是否满足某特定状态或条件。

(4) 测试用例分析。使用此功能对测试数据或用例进行效率分析,从而可优化测试。

**3. 产品应用**

单元级软件测试已经被公认为行之有效的软件测试方法,使用单元级软件测试可在软件开发早期发现软件故障或缺陷,从而提高软件可靠性同时减少软件测试开销。传统的单元级软件测试采用人工方式编写测试驱动与桩模块,具有测试程序可靠性低、开销大、依赖于测试人员经验等问题。同时,由于大多测试时间花费在编写测试程序上,测试人员积极性不高,给软件测试效果带来影响。

有鉴于此,LDRA 公司成功开发出单元级测试工具 Testbed/TBrun,该工具功能全面、易于使用,支持 Windows、UNIX、Linux 和麒麟等操作系统,不仅适合于主机平台软件测试,同时适合于嵌入式软件测试,已成功地应用于国内各大研究机构和软件测试部门。使用 Testbed/TBrun 可自动产生软件测试驱动、桩模块,节省时间,测试人员可将重点放在设计测试用例上,提高软件测试效率,同时提高软件测试人员积极性。LDRA Testbed 产生的静态、动态结果均可以图形化显示,直观方便,支持的图形化显示功能有柱状图、流程图、调用图、Kiviat 图。

### 11.3.5 CodeTest

**1. 工具概述**

CodeTest 是专为嵌入式系统设计的软件测试工具，CodeTest 为追踪嵌入式应用程序、分析软件性能、测试软件的覆盖率以及内存的动态分配等提供了一套实时在线的高效率解决方案。CodeTest 还可以通过网络远程检测被测系统的运行状态，可以满足不同类型的测试环境，给整个开发和测试团队带来高品质的测试手段。CodeTest 可以支持几乎所有的主流的嵌入式软件和硬件平台，可以支持多种 CPU 类型和嵌入式操作系统。CodeTest 可支持几乎所有的 32/64 位 CPU 和部分 16 位 MCU，支持的数据采集时钟频率高达 133MHz。CodeTest 可通过 PCI/cPCI/VME 总线采集测试数据，也可通过 MICTOR 插头、飞线等手段对嵌入式系统进行在线测试，无需改动被测系统的设计，CodeTest 与被测系统的连接方式灵活多样。

**2. 产品功能**

CodeTest 包括以下 3 个产品，分别用于嵌入式软件系统开发的不同阶段的测试，可满足不同应用的需求。

(1) CodeTest Native。在早期的开发阶段，采用 CodeTest Native 的插桩器可以实现较快的软件测试和分析。虽然此阶段的测试和分析不是实时测试，但这是没有目标硬件连接时分析和查找问题的最好方法。采用 CodeTest，可以通过提高软件测试的代码覆盖率、查找和分析内存的泄漏和深度追踪来确保软件的正常运行。

(2) CodeTest SWIC (software in circuit)。当有硬件连接到测试系统时，可以采用目标硬件工具。一般说来，在这一阶段，逻辑分析仪、仿真器和纯软件工具可以用来确定系统是否正常工作，但是采用这些测试工具往往增加了工程师工作的难度和压力。而采用 CodeTest SWIC，通过目标代理来测试和分析目标硬件，无需使用硬件工具。CodeTest SWIC 插桩器还可以很方便地让用户从 CodeTest Native 的 desktop-stimulated 测试跳转到目标硬件的实时测试。跳转后，插桩器、脚本的文件格式和数据不受 Native 环境影响。对于大多数开发者，CodeTest 可以大大节约开发的时间。虽然 CodeTest SWIC 工具不提供外部硬件测试系统的细节情况，但其为硬件测试难题提供了解决方案，具有强大的代码覆盖分析、内存分析和追踪分析功能，且在真实硬件环境中运行，价格低廉。

(3) CodeTest HWIC (hardware in circuit)。当进入此阶段时，会需要一组能提供监视软件测试深度和精确度的工具链。CodeTest HWIC 工具采用外部硬件辅助和相应的通信系统来实现最大程度的软件实时测试。与逻辑分析仪和仿真器不同，CodeTest HWIC 具有处理复杂嵌入式系统的实时测试能力。CodeTest 外置探测的硬件系统主要包括控制和数据处理器、大容量内存和可编程的升级定时器，因此大型测试的时间精度可在±50ns 内。CodeTest HWIC 除了提供测试代码覆盖率分析、内存分析和追踪分析，其精确的实时测试能力还可以帮你查出软件性能和质量上的问题所在。

CodeTest 系统包括以下 4 个功能模块。

(1) 性能分析。CodeTest 能够同时对多达 128000 个函数进行非采样性测试，精确计算出每个函数或任务(基于 RTOS 下)的执行时间或间隔，并能够列出其最大和最小的执行

时间。CodeTest 的性能分析功能也能够为嵌入式应用程序的优化提供依据，使软件工程师可以有针对性地优化某些关键的函数或模块，从而改善整个软件的总体性能。

（2）覆盖分析。CodeTest 提供程序总体概况、函数级代码以及源级覆盖趋势等多种模式来观测软件的覆盖情况。CodeTest 覆盖率信息包括程序实际执行的所有内容，而不是采样的结果，它以不同颜色区分运行和未运行的代码，CodeTest 可以跟踪超过一百万个分支点，特别适用于测试大型嵌入式软件。

（3）动态存储器分配分析。CodeTest 能够显示有多少字节的存储器被分配给了程序的哪一个函数，统计出所有的内存的分配情况，指出存储体分配的错误，让测试者可以同时看到其对应的源程序内容。

（4）追踪分析。CodeTest 可以按源程序、控制流、高级模式来追踪嵌入式软件，最大追踪深度可达 150 万条源级程序。其中，高级模式显示的是 RTOS 的事件和函数的进入退出，给测试者一个程序流程的大框图；控制流追踪增加了可执行函数中每一条分支语句的显示；源程序追踪则又增加了对被执行的全部语句的显示。在以上三种模式下，均会显示详细的内存分配情况，包括在哪个代码文件的哪一行，哪一个函数调用了内存的分配或释放函数，被分配的内存的大小和指针，被释放的内存的指针，出现的内存错误等。

### 3. 产品应用

CodeTest 可同时进行软件性能、代码覆盖率、动态内存分配的分析。该工具套件非常适合嵌入式软件测试，其测试具有很高的可靠性。CodeTest 新推出的 VME 能帮助软件开发人员集中精力于软件代码设计，不必费神于硬件、目标系统的连接以及测试方案等。CodeTest-VME 测试卡采用 VME 总线连接工业标准，通过总线分析器提取目标板的详细信息，支持所有的 CodeTest 功能，支持外界电源、96 脚背板连接方式、单槽 6U、160MM、以太网连接、AUI 带适配器方式，板级测试时可支持自检，可远程控制。

该产品目前支持 CPU：PowerPC、ColdFire、ARM、x86、MIPS、DSP（TI、ADI、Starcore）等；操作系统：VxWorks、AE、OSE、QNX、pSOS、Chorus、Lynux、Win CE、Linux、麒麟等；总线：PCI、cPCI、PMC、VME 等；处理器：29K、68K、ARM、Coldfire、H8、i960、MIPS、MPC8xx、PowerPC、SH、SPARC、X86；操作平台：Windows 95/NT、UNIX。

## 11.3.6　QuestaSim

### 1. 概述

QuestaSim 是 Mentor Graphics 公司推出的用于可编程逻辑器件仿真测试的工具，能够对被测设计进行功能仿真、门级仿真和时序仿真，是目前应用最为广泛的 FPGA 仿真测试工具。

### 2. 功能

QustaSim 是 FPGA 仿真工具 ModelSim 的升级版，相比 ModelSim 工具，其在仿真性能、覆盖率、界面优化、运行效率和稳定性等方面均有较大提升，主要功能和优点如下：

（1）支持主流芯片厂商如 Altera、Xilinx、Actel 等的芯片类型，并能够与厂商提供的 EDA 工具方便地集成，便于开发过程中进行仿真验证和设计迭代；

(2) 支持大规模 FPGA 设计的仿真，仿真速度优于其他的仿真工具；

(3) 能够收集功能、条件、分支、语句、条件、状态机、翻转、断言等多种类型的覆盖率信息，便于评估仿真测试的充分性，同时支持覆盖率的累计、合并和未覆盖点的分析，并能够生成数据库、网页或文本类型的覆盖率统计信息，通过关联测试覆盖率和测试对象，促进验证进度跟踪和测试资源的高效配置；

(4) 支持 OVM、UVM 等高级验证方法学；

(5) 支持主流 RTL 语言如 VHDL、Verilog、System Verilog 等，支持混合语言设计的复杂 FPGA 设计的仿真验证；

(6) 能够辅助搭建测试平台的框架，提高测试平台的编写效率；

(7) 能够支持复杂激励信号组合的自动生成，激励情形以约束的形式描述，极大地缩短测试开发和修改的次数，提高测试设计的效率；

(8) 均有测试分析能力，将原始覆盖率数据转变为可操作、易识别的信息，能够识别冗余测试，并基于测试项覆盖率判断是否进行优先测试；

(9) 支持多种验证引擎，并最大化地发挥断言的技术优势，有效提升验证和调试效率，保证设计质量并改善验证结果的可预测性和可观测性，便于查找错误的源头，提高测试的效率。

**3. 产品应用**

QuestaSim 主要用于 FPGA 的功能仿真、时序仿真和门级仿真，既可作为开发人员的设计调试工具，又可作为验证人员的验证辅助工具，极大地提高设计和开发的效率。

### 11.3.7 其他动态测试工具

其他动态测试工具有：

(1) Compuware 公司的 DevPartner，支持 C++、Java、VisualBasic，包含代码覆盖率分析工具 TrueCoverage、代码效率分析工具 TrueTime 和内存分析检查工具 BoundsChecker；

(2) IBM 公司的 Rational PurifyPlus，支持 Java、C/C++、VisualBasic 和.NET，包含代码覆盖率分析工具 pureCoverage、代码效率分析工具 pureQuantity 和内存检查工具 purify。

## 11.4 测试管理工具

### 11.4.1 TestCenter

**1. 工具概述**

TestCenter 测试管理工具是一款功能强大的测试管理工具。TestCenter 采用面向需求的测试而不是面向操作流程的测试。TestCenter 采用针对用户业务流程的测试，支持顺序流程，同时支持工作流的操作。使用 TestCenter，可以实现测试用例的过程管理，对测试需求过程、测试用例设计过程、业务组件设计实现过程等整个测试过程进行管理；可以实现测试用例的标准化，即每个测试人员都能够理解并使用标准化后的测试用例，降低了测试用

例对人的依赖。TestCenter 提供测试用例复用，用例和脚本都能够被复用，以保护测试人员的资产；提供可伸缩的测试执行框架，提供自动测试支持；提供测试数据管理，帮助用户统一管理测试数据，降低测试数据和测试脚本之间的耦合度。

**2. 产品功能**

TestCenter 是面向测试流程和测试用例库的测试管理工具，具有以下功能。

(1) 测试需求管理。TestCenter 支持对测试需求的全方位管理：支持 Word、Excel 格式的测试需求导入；支持需求条目化；支持测试需求评审；支持测试需求与用例的关联；支持测试需求树，树的每个节点是一个具体的需求，也可以定义子节点作为子需求，每个需求节点都可以对应到一个或者多个测试用例。

(2) 测试用例管理。测试用例允许建立测试主题，通过测试主题来过滤测试用例的范围，实现有效的测试。支持手工测试用例和自动化测试用例，支持测试用例树型结构。支持测试用例的各种状态：执行通过、未执行、执行失败；支持测试用例关联缺陷；支持测试关联到需求。支持执行中的测试用例管理。实现测试用例的标准化即每个测试人员都能够理解并使用标准化后的测试用例，降低了测试用例对个人的依赖。

(3) 测试业务组件管理。支持软件测试用例与业务组件之间的关系管理，通过测试业务组件和数据搭建测试用例，实现了测试用例的高度可配置和可维护性。

(4) 测试流程管理。管理测试中的流程，测试需求创建、测试需求评审、测试计划、测试执行、缺陷管理等流程。支持测试计划管理、测试计划多次执行；测试需求范围定义、测试集定义。支持测试自动执行(通过调用测试工具)；支持在测试出错的情况下执行错误处理脚本，保证出错后的测试用例脚本能够继续被执行。

(5) 自动测试框架。支持存放自动测试脚本；支持不同类型的自动测试工具；支持配置化的自动测试用例；支持自动测试框架，支持测试执行中的数据管理；支持自动测试日志。自动测试框架能够极大地简化了自动测试过程和用例。

(6) 测试结果日志查看。具有截取屏幕的日志查看功能。

(7) 测试结果分析。TestCenter 中带有功能强大而全面的报表系统，采用集成化的用户界面，每一次测试对应一个 TestID。TestCenter 报表的功能是对测试过的案例，返回一个测试结果，包括有正确返回(与预期相同)与错误返回(与预期不同，并且同时返回错误时候的状态：屏幕画面、设备值等)；TestCenter 的报表可指定错误目录，生成测试报告，生成异常错误数据和报告；其统计功能包括有 3 种测试状态(未测试、成功、失败)的百分比、针对测试案例的百分比；测试针对测试案例生成测试日志，截取所有的屏幕界面、以 HTML 的格式存放，可以通过链接直接访问。支持多种统计图标，比如需求覆盖率图、测试用例完成的比例分析图、业务组件覆盖比例图等。

(8) 缺陷管理。提供了最好用的缺陷管理模块。支持缺陷流程管理，用户可以自定义缺陷流程；支持缺陷属性自定义；支持自定义的缺陷报表和缺陷分析。支持从测试错误到曲线的自动添加与手工添加；支持自定义错误状态、自定义工作流的缺陷管理过程。

**3. 产品应用**

该产品安装简单、使用方便，支持 Windows 操作系统和 SQL Server 数据库。C/S 版本的 TestCenter 也可以连接该公司自己研发的缺陷管理系统，与其他同类工具相比具有以下

特点：

(1) 中文界面，更容易使用和理解；

(2) 可自定义工作流，适应各个公司的具体情况；

(3) 强大的报表分析系统，可根据用户要求统计出 BUG 的各种情况以及各类对比；

(4) 操作简易，使用过滤器搜索 BUG 简便。

## 11.4.2 TP-Manager

### 1. 工具概述

TP-Manager 支持软件测试各个阶段的工作，符合主要测试标准或规范的管理要求，能够自动化实现软件测试过程的规范化管理，有效提高测试过程管理的工作效率。该工具具备以下特点：

(1) 能够有效规范测试流程，引导评测人员严格按照软件测试规范进行各阶段的测试活动；

(2) 通过底层数据库对测试项目信息、测试各阶段活动、测试结果及测试数据等信息进行合理化管理，保证了所有信息的完整性和一致性；

(3) 能够自动进行测试信息的分类、统计，辅助测试人员对繁杂的测试设计及结果信息进行合理化分析；

(4) 能够自动化生成一整套规范的软件测试文档；

(5) 通过联网作业方式为软件测试小组成员创建一体化协同工作平台。

TP-Manager 以向导服务方式引导测试人员严格按照软件测试各阶段的要求开展软件测试工作，包括清晰明确地梳理测试需求，基于测试需求制定测试计划，按照测试计划设计测试用例，遵循测试用例设计执行软件测试，依据客观测试结果自动分析归纳测试结论；该工具可以建立并维护测试数据库，通过联网作业方式为软件测试小组成员创建一体化协同工作平台，确保数据和信息的完整性和一致性；该工具能够自动进行测试信息的分类、统计和分析，自动生成规范化的软件测试文档。

### 2. 产品功能

TP-Manager 提供一个软件测试过程管理向导树，包括项目基本信息管理、软件测试需求分析、测试策划、测试设计、测试执行、测试总结等多层次的工作向导。软件测试人员只要按照向导树规定的步骤完成每一步的工作，就能逐项落实软件测试的各项管理要求和技术要求。TP-Manager 主界面如图 11-1 所示。下面介绍该工具为用户提供的诸多引导服务功能。

1) 软件测试项目基本信息管理

工具提供软件测试项目基本信息管理向导，包括对项目基本信息、测试类别、测试用例设计方法、软件问题类别、软件问题级别、测试项的优先级、问题标识结构等管理向导。

为了统一软件测试项目测试类型的划分方法，工具为用户提供了定制软件测试类型术语的功能，用户可根据需要定制测试类型。定制的测试类型可用于软件测试需求分析、测试策划阶段对测试对象测试要求的层次化分解。如，用户订制测试类型为功能测试、性能测试、接口测试、人机交互界面测试、安装性测试等，定制了这些类型之后，在软件测试需求分



图 11-1 TP-Manager 主界面

析、测试策划阶段就可以直接在定制的测试类型集合中选择这些测试类型。该功能具有两个方面的目的，一是便于统一软件测试项目的测试类型划分方法，二是省略了用户多次输入测试类型的操作。

在软件测试实践中，存在着多种软件测试用例的设计方法，如有效类、无效类、边界值、压力、时序、猜错法等。为了统一测试用例设计方法，工具为用户提供了定制测试用例设计方法的功能，通过这个功能统一定义测试项目的测试用例设计方法，例如可把测试用例设计方法定制为有效类、无效类、边界值、压力、时序或者等价类划分、错误猜错法、边界值分析法等。定制的这些方法将用于测试用例设计选择测试用例的设计方法。

工具提供对于每个测试项按照测试用例设计方法分类统计功能，便于测试管理人员分析测试用例设计的充分性。

软件问题是指在软件测试期间所发现的被测软件存在的问题，该工具以软件问题报告的形式提交被测软件存在的问题。对于软件问题的类别和级别，不同的用户有不同的分类、分级方法。工具为用户提供了软件问题类别和软件问题级别的定制功能，可以为某个测试项目单独定制软件问题类别和软件问题级别，如将软件问题的类别定义为软件任务书问题、需求分析问题、软件设计问题、软件编码问题、其他问题等，将软件问题的级别定义为致命问题、严重问题、一般问题、轻微问题等等。

用户定制的软件问题级别和软件问题类别将用于软件问题报告，当测试人员发现被测软件的问题时，通过工具提交软件问题报告，软件问题报告中的问题级别和问题类别只能在定制的类别和级别中选择。如果需要调整软件问题的类别和级别，用户可以在该工作界面重新定制，重新定制之后，工具将自动并且全部刷新所有的问题报告中的相关信息。

在《软件测试需求规格说明》和《软件测试计划》中，可以为软件测试项定义测试项的优

先级。不同的用户，甚至同一用户对于不同软件测试项目存在着多种软件测试项的优先级的表述方式，如将优先级表示为A、B、C等级，或者1、2、3等级等。工具为用户提供了定制软件测试项优先级表述方式的定制功能，用户可以按照自己的规范进行自我定制。用户定制的软件测试项优先级表述方式，将用于《软件测试需求规格说明》和《软件测试计划》中测试项优先级的选择，对于软件测试需求分析阶段或软件测试策划阶段分解出的测试项，其优先级只能在已经定制的测试项优先级中选择。如果需要调整测试项的优先级，用户可以在该工作界面重新定制，重新定制之后，工具将自动并且全部刷新所有的测试项的优先级。

软件问题报告中有软件问题标识的栏目。不同的用户，甚至同一用户对于不同软件测试项目存在着多种软件问题标识的表述方式。工具为用户提供了定制软件问题标识表述方式的定制功能，用户可以按照自己的规范进行自我定制。工具为用户提供了最多4个字段的定制功能，在每个字段中，用户可以定义多个字符串。

用户定制的软件问题标识表述方式，将用于提交软件问题报告时生成软件问题标识。

工具自动对同一标识的软件问题分类计数，按照提交问题的先后顺序，给同一标识的软件问题自动编号。

2）软件测试需求分析的向导与管理

软件测试必须具有明确的测试依据。软件测试依据可包括软件测试合同、软件测试技术协议、软件测试任务书、被测软件的需求规格说明、被测软件的接口需求规格说明、被测软件的设计说明（主要用于部件测试）、被测软件的用户手册等。

梳理测试依据功能为用户提供了逐层分解测试依据的工作界面，便于测试人员清晰、全面地梳理测试依据。测试人员应尽可能将软件测试依据分解细化，并明确地描述测试依据的出处及其说明，如“需求分析中2.1.1.1节的打印队列管理功能”。

测试人员梳理出来的测试依据将用于建立软件测试项与测试依据的追踪关系，工具自动查询软件测试项与测试依据的追踪关系，建立测试依据与测试项的追踪关系表；如果某项测试依据没有被测试项所追踪的话，工具将自动给出红色的报警提示。

3）软件测试策划阶段的向导与管理

对于大多数测试组织来讲，软件测试策划阶段是软件测试项目的第一个阶段（工具支持向导功能的定制，可隐藏测试需求分析阶段，工具直接进入测试策划阶段）。

软件测试策划的管理工作包括《软件测试计划》文档基本信息管理、设计软件测试环境、规划软件测试策略、安排测试的组织与人员、分析确定被测对象、针对每个被测对象分解测试项、确定测试项目的终止条件、输出《软件测试计划》文档等工作。

如果软件测试组织在软件测试策划阶段之前已经使用该工具进行了软件测试需求分析，那么，在软件测试策划阶段将直接沿用需求分析阶段所识别的被测对象、被测软件概述信息、针对每个被测对象所定义的测试类型和各个测试项、测试项目终止条件等信息。

用户在软件测试需求分析阶段或者在软件测试策划阶段对于被测对象、被测软件概述信息、针对每个被测对象所定义的测试类型和各个测试项、测试项目终止条件等信息的更改，将直接导致对《软件测试需求规格说明》和《软件测试计划》的同步更改，这一功能有效地保障了《软件测试需求规格说明》和《软件测试计划》的一致性。

4）测试设计与实现阶段的向导与管理

测试设计与实现阶段的主要任务有两个，一是遵循软件测试计划（或软件测试需求规格

说明)设计软件测试用例，二是实现支持测试用例所需的测试软件或者测试工具。该工具侧重于支持软件测试用例的设计及其管理，对于与测试用例相关联的测试软件、测试工具、输入域中的数据或图片、期望域中的数据或图片提供了附件管理功能，从而使得软件测试用例及其相关附件紧密耦合，管理清晰。

按照软件测试标准或规范的基本要求，测试用例的设计必须遵循测试计划，即必须在测试计划的框架下开展测试用例的设计工作。工具提供遵循软件测试计划设计测试用例的功能，在软件测试策划时分析定义的被测对象、测试类型、测试项都将作为测试用例设计的约束性框架，同时也是引导框架。工具能够确保被测对象、测试类型、测试项的完整性及其层次关系，测试人员只需在各个软件测试项下设计测试用例即可。

5）测试执行阶段的向导与管理

按照软件测试标准和规范的要求，原则上应依照《软件测试说明》中描述的测试用例开展测试用例的执行工作，在实际测试期间，如果需要增加、删除、变更测试用例，应该得到相应的控制，并且必须保持测试说明与测试记录的一一映射关系。

在测试执行期间，如果某些(或某个)测试用例没有完整执行或没有执行，测试人员应分别针对每个未完整执行(包括未执行)的测试用例说明原因。

本工具支持软件测试标准和规范的上述要求，严格、有序地建立了测试说明与测试记录的一一映射关系，实现了测试记录与测试说明相同栏目的同步更改，确保了测试说明与测试记录的一致性。

6）测试总结阶段

根据软件测试标准和规范的要求，必须基于客观翔实的软件测试记录进行软件测试总结。工具通过对测试数据库的查询，针对不同的被测对象，用统计表的形式分别统计出测试用例的设计数目、完整执行的数目、部分执行的数目、未执行的数目。

工具通过对测试数据库的查询，针对不同的被测对象，用统计报表的形式分别给出测试用例的执行状态、执行结果、对应的问题单等信息。工具的自动查询、统计功能，确保了统计报表的客观性和准确性。在统计报表中还提供了测试用例的链接和软件问题报告单的链接功能，当测试人员需要查看某个测试用例、某个软件问题报告单时，可以通过链接功能快速查阅。

工具通过对测试数据库的查询，针对不同的被测对象，用统计报表的形式分别统计出未完整执行(包括未执行)的测试用例，测试人员可在此统计报表中填写未完整执行的原因，当用户需要查看某个测试用例时，该统计报表中还提供了测试用例的链接功能，便于测试人员快速查看相应的测试用例。

用户提交的测试相关信息均保存在本测试项目的数据库中，工具将按照用户定制的文档模板的格式要求，自动生成测试文档。

**3. 产品应用**

该产品的安装和运行需满足以下要求：计算机系统内容大于 256MB，可用磁盘空间大于 2GB、Word2003 及以上文字处理系统、Access2003 及以上数据库。TP-Manager 是在 .net 的平台上开发的，软件的安装和运行需要 .net 2.0 FrameWork 的支持。如果计算机上没有安装 .net，安装程序将会自动在计算机上安装 .net FrameWork。

### 11.4.3 其他测试工具

其他测试管理工具有：TestDirector（QualityCenter）、QADirector、Certify、Product manager、SilkCentral Test Manager、Doors、E-manager、Testmanager、TestView Manager、Professional 等。

软件测试工具是实现软件测试技术的软件产品，对于多数软件测试来说都是必不可少的。本章对常用的静态测试工具、动态测试工具和测试管理工具进行了介绍，这些工具对于发现软件缺陷、提高测试及管理效率发挥了重要作用。但是，需认识到，测试工具也有自身的局限性，软件测试工具不能取代手工测试，手工测试可以比自动测试发现更多的缺陷。目前，各种测试工具种类繁多，在选用工具时应综合考虑工具的功能、价格、售后服务等因素，针对不同开发语言、不同应用领域、在软件工程的不同阶段选择合适的测试工具，只有这样才能充分发挥测试工具的作用，提高软件测试效率和软件质量。

CHAPTER 12

第 12 章

# 软件测试文档

不论哪一种测试级别，基本的测试活动都包括：测试策划、测试设计与实现、测试执行和测试总结。在每个测试活动中，都会编写相应的软件测试文档，本章将说明测试文档的内容，给出文档模板，并说明各文档的常见问题。

## 12.1 概述

目前，在软件研制的过程中对软件测试的策划不够重视，导致测试环境保障有限、测试不充分等问题的发生。解决这些问题关键是提高对软件测试重要作用的认识，并切实制定和落实相关措施。这些措施主要包括以下内容。

（1）尽早地开展测试策划工作。测试策划的工作产品是软件测试计划。由于测试计划中需要提出测试资源、人员、测试环境和测试策略等较为具体的要求，因此有利于保证测试资源、人员、测试方法和测试环境的有效落实。同时，测试计划对测试工作进入条件、时间进度等都有较为明确的要求，也有利于促进软件研制人员对测试工作提高认识。

（2）尽早地开展测试。尽早地开始测试是指在软件研制的早期，例如需求分析、设计和编码时就开始测试，与开发人员一起进行文档评审、代码审查、代码走查等。同时，也开展测试用例的设计、测试环境的建立以及测试程序的开发等工作。

（3）大型项目成立独立的测试组。测试对人员的专业技能要求较高，不仅需要测试技术，也需要有一定的开发能力。在许多软件研制机构中，没有设置专门的测试部门，测试工作由开发人员自己完成。由于开发人员测试自己开发的软件一般都会存在惯性思维，限制了从不同角度考虑问题，使用方面的测试也只能按照自己的习惯。因此，对大型项目来说，成立独立的测试组能够较好地解决此类问题。

（4）加强测试度量工作。测试的充分性如何保障是测试需要面对和解决的问题。加强测试度量工作可以一定程度上为分析测试充分性提供数据基础。与测试活动相关的度量主要包括软件源代码规模、测试项数、测试用例数、测试用例执行数量、缺陷数、缺陷密度、不同问题等级数、不同问题类型数、测试工作量、问题被发现的阶段和测试覆盖率等。通过对这些基本测量数据的分析，可以获得对软件质量、测试工作情况的基本分析结果，对进一步提高软件质量和测试工作效率都是非常有意义的。

（5）深入开展缺陷分析工作。深入开展缺陷分析对软件开发和测试都可以起到促进作用。建立基本的缺陷数据库，收集与缺陷相关的信息对缺陷分析至关重要。与缺陷相关的

信息包括缺陷描述、级别、类型、缺陷引入活动、与之相关的需求、与之相关的测试用例、软件名称、测试类型、测试执行时间、负责人、缺陷是否更动、缺陷更动内容等。通过对上述信息的分析，可以发现软件研发过程中哪些活动最容易出现问题，测试应重点关注的内容等。

上面这些措施的落实和体现形式就是软件测试文档，通过按照要求执行测试活动，编写测试文档可以反映项目是否按要求执行各种测试活动，进而有效地控制测试过程，保证测试活动的有效性和充分性。

## 12.2 制定测试计划

软件测试计划是开展软件测试工作的基础。因此，应在软件研制初期就应制定顶层的软件测试计划，主要是提出测试策略、测试人员、测试资源、测试通过准则和测试进度安排等，并随着软件研制活动的逐步开展，逐步细化测试计划，制定每个测试级别的详细测试计划。

### 12.2.1 测试计划内容

制定测试计划的活动内容包括以下内容。

(1) 按照测试依据和软件质量要求确定软件测试的需求。

① 梳理软件需求，明确需要测试的范围。

② 说明测试的总体要求，包括测试级别、测试类型、测试策略等。

③ 定义测试项，每个测试项需要明确的内容包括：确定每个测试项的名称和标识、说明每个测试项的具体测试要求、确定每个测试项的测试方法、说明对每个测试项进行测试时所需要的约束条件、确定每个测试项通过测试的评判标准、提出对每个测试项进行测试用例设计时所需要考虑的测试充分性要求、规定完成每个测试项测试的终止条件、定义每个测试项目的测试优先级，优先级一般可以根据依据文件中定义的相应需求的优先级进行定义、建立每个测试项与测试依据之间的追踪关系。

④ 制定测试策略，包括测试数据生成策略、测试信息注入与捕获方法、测试结果分析方法等。

(2) 分析测试环境需求，包括计算机硬件、接口设备、计算机操作系统、支持软件、专用测试软件、测试工具和测试数据等。

(3) 提出测试人员安排。一般情况下，单元测试和集成测试可由开发人员完成，配置项测试、系统测试由专门的测试人员完成。

(4) 安排测试的进度计划。应依据软件研制进度、测试需求、测试环境、人员等情况，制定合理可行的软件测试进度计划。

(5) 制定测试通过的准则。单元测试通过的准则示例如下：

① 软件实现与设计文档一致；

② 语句和分支覆盖率达到 100%，如果确实无法覆盖应进行分析，并说明未覆盖的原因；

③ 代码审查中强制类错误都得到解决；
④ 单元测试发现的问题得到修改并通过回归测试；
⑤ 单元测试报告通过评审。
（6）分析测试活动中可能存在的风险，并制定相应的缓解和应急计划。

## 12.2.2　测试计划模板

1　范围
1.1　标识
a）本文档标识：
b）标题：
c）本文档适用的范围：
d）本文档的版本号。
1.2　文档概述
1.3　被测软件概述
概述被测软件的下列内容：
a）被测软件的名称、版本、用途；
b）被测软件的组成、功能、性能和接口；
c）被测软件的运行环境。
1.4　与其他文档的关系
2　引用文档

**表 2-1　引用文件表**

| 序号 | 引用文档标题 | 引用文档标识 | 编写单位 | 出版日期 |
|---|---|---|---|---|
| | | | | |

3　术语和定义
本章给出所有在本文档中出现的专用术语和缩略语的确切定义。

**表 3-1　术语和缩略语表**

| 序号 | 术语和缩略语名称 | 术语和缩略语说明 |
|---|---|---|
| | | |

4　测试内容与方法
4.1　测试总体要求
根据被测软件研制要求或系统研制要求、软件需求规格说明及其他等效文档，结合被测软件的级别及其质量要求，提出测试的范围、测试级别、测试类型、测试策略等总体要求。
测试级别分为：单元测试、集成测试、配置项测试和系统测试。集成测试可分为：配置项集成和系统集成。

4.2 测试项及测试方法

4.2.X (被测对象)

4.2.X.Y (被测对象的测试类型)

4.2.X.Y.Z (被测对象的测试项)

| 测试项名称 | | 测试项标识 | |
|---|---|---|---|
| 测试项说明 | | | |
| 测试方法 | | | |
| 约束条件 | | | |
| 评判标准 | | | |
| 测试充分性要求 | | | |
| 测试项终止条件 | | | |
| 优先级 | | | |
| 追踪关系 | | | |

4.3 软件问题类别及严重性等级

**表 4-1 问题类别表**

| 序号 | 问题类别名称 | 问题类别说明 |
|---|---|---|
| 1 | 程序错误 | 运行程序与相应的文档不一致,而文档是正确的 |
| 2 | 文档错误 | 运行程序与相应的文档不一致,而程序是正确的 |
| 3 | 设计错误 | 虽然运行程序与相应的文档一致,但是存在设计缺陷,可能产生错误 |
| 4 | 其他错误 | 非程序错误、文档错误、设计错误的错误类别 |

**表 4-2 问题严重性等级**

| 序号 | 问题级别名称 | 问题级别说明 |
|---|---|---|
| 1 | 1 级 | (1) 有碍于运行或任务的基本能力的实现;<br>(2) 危害安全性、保密性或其他关键性要求 |
| 2 | 2 级 | (1) 对运行或任务的基本能力产生不利影响且没有变通的解决方案;<br>(2) 对项目的技术、费用、进度风险或对系统寿命期的支持产生不利影响,且没有变通的解决方案 |
| 3 | 3 级 | (1) 对运行或任务的基本能力产生不利影响,但存在变通的解决方案;<br>(2) 对项目的技术、费用、进度风险或对系统寿命期的支持产生不利影响,但存在变通的解决方案 |
| 4 | 4 级 | (1) 给用户/操作员带来不便或烦恼;但不影响运行或任务的基本能力;<br>(2) 给开发或支持人员带来不便,但不妨碍工作的完成;<br>(3) 任何其他影响 |

5　测试环境

5.1　软硬件环境

对此次测试所需的软硬件环境进行描述。

a）整体结构。描述测试工作所需的软硬件环境的整体结构，例如需建立的网络环境，还需描述网络的拓扑结构和配置。

b）软硬件资源。描述测试工作所需的系统软件、支撑软件以及测试工具等，包括每个软件项的名称、版本、用途等信息；描述测试工作所需的计算机硬件、接口设备和固件项等，包括每个硬件设备的名称、配置、用途等信息。

**表 5-1　测试资源配置表**

| 序号 | 资源名称 | 配置 | 数量 | 用途 | 维护人 |
|---|---|---|---|---|---|
| | | | | | |

5.2　测试场所

描述执行测试工作所需场所的地点、面积以及安全保密措施等。

5.3　测试数据

描述测试工作所需的真实或模拟数据，包括数据的规格和数量等。

5.4　环境差异影响分析

描述软硬件环境及其结构、场所、数据与被测软件研制要求或系统研制要求、软件需求规格说明及其他等效文档要求的软硬件环境、使用场所、数据之间的差异，并分析环境差异可能对测试结果产生的影响。

6　测试进度

描述主要测试活动的时间节点、提交的工作产品和人员安排等。

**表 6-1　测试进度安排**

| 序号 | 工作内容 | 开始时间 | 结束时间 | 工作产品 | 人员 |
|---|---|---|---|---|---|
| | | | | | |

7　测试结束条件

描述测试结束的条件，包括正常和异常结束的条件。

8　软件质量评价内容和方法

描述基于此次测试的软件质量评价内容和评价方法。

9　测试通过准则

描述被测软件通过此次测试的准则。例如：可能允许的遗留问题的级别、潜在缺陷密度等。

10　测试人员组成

**表 10-1　测试人员组成表**

| 序号 | 角色 | 姓名 | 职称 | 主要职责 |
|---|---|---|---|---|
| | | | | |

11 测试数据记录、整理和分析

描述根据本计划实施测试时，获得测试结果数据的整理和分析过程，说明达到测试目标的要求和测试充分性分析方法。对未通过的测试用例，要特别说明未通过的原因和需要采取的措施。

12 测试风险分析

从时间、技术、人员、环境、分包、项目管理等方面对完成此次测试的风险进行分析，并提出应对措施。

13 追踪关系

**表 13-1 测试依据与测试项的追踪关系表**

| 序号 | 测试依据标识 | 测试依据 | 测试项标识 | 测试项名称或未追踪原因说明 |
|---|---|---|---|---|
| | | | | |

**表 13-2 测试项与测试依据的追踪关系表**

| 序号 | 测试项标识 | 测试项名称 | 测试依据标识 | 测试依据 |
|---|---|---|---|---|
| | | | | |

## 12.2.3 测试计划常见问题

软件测试计划的制定需要与研制人员、项目管理人员充分地沟通和协调，保证测试范围、测试方法、测试资源和测试进度的有效落实。在制定软件测试计划中常见的问题如下。

(1) 对被测软件的描述不完整。存在缺少被测软件版本、规模、关键等级的信息，运行环境中缺少关键的硬件配置信息和相关软件环境的描述，接口描述不清晰等问题。对被测软件的描述应包括：

① 被测软件的名称、版本、规模、关键等级；

② 运行环境应包括软/硬件环境和网络环境等，如果有数据库系统还应描述数据库系统的信息；

③ 主要功能、性能和接口，接口描述建议采用图形化方式进行清晰地描述。

(2) 引用文件描述不全面。在引用文件描述中缺少软件研制、测试所需要遵循的标准和规范，缺少被测软件相关技术文件。引用文件应包括：

① 软件开发和测试应遵循的标准和规范；

② 被测软件相关文档，例如软件评测任务书、软件需求规格说明书、用户手册等，需要根据测试级别确定被测软件的相关文档；

③ 测试中需要遵循或依据的文件，例如通信协议，与测试活动相关的会议纪要等。

(3) 测试总体要求中提出的测试类型不全面，与测试任务要求的不一致，未说明测试仿真环境的总体设计要求等。

（4）测试项定义的不完整、不具体，主要体现在以下两个方面：

① 对测试需求覆盖的不全面，例如缺少对安全性需求的测试，缺少对工作模式的测试，缺少对隐含需求的测试等；

② 对每个测试项说明的不具体、不完整，主要表现在：测试项说明不具体，对需要测试内容描述不具体，特别是性能、精度等有具体数值要求的测试内容没有详细说明，对评估其满足情况的允许偏差未进行说明；测试方法不具体，主要表现在未说明测试数据的注入方式、测试结果的捕获方法以及测试结果分析方法等；测试方法不恰当，主要表现在测试方法无法满足测试要求，例如对毫秒级性能测试要求，应使用更精确的测量方法进行测试；缺少测试项约束条件的描述；缺少测试项评判标准的描述，特别是性能测试项的评判标准、应满足的误差要求未进行具体说明；测试充分性要求不具体，主要表现在未对测试用例设计充分性方面提出具体要求；测试项终止条件不恰当，特别是容量、强度等的测试项终止条件未根据测试项的特点进行定义；优先级未定义或定义的不恰当，优先级定义不恰当的最突出表现是所有测试项的优先级都相同；缺少测试项对测试依据之间的追踪关系或追踪关系不正确。

（5）软/硬件环境描述不全面，不详细。测试的软/硬件环境直接影响测试结果，因此需要对测试环境进行全面、详细地描述，以便保证测试环境的有效性。存在的问题主要表现如下：

① 硬件环境不准确，被测软件的运行环境与实际运行环境不一致；

② 测试环境考虑的不全面，例如强度测试需要的测试环境要求更高，考虑不全面时可能造成强度测试无法实现；

③ 测试所需软件的要求不具体，例如对测试程序所需要实现的功能、性能未提出要求，影响测试用例的实现；

④ 硬件环境的配置，测试软件的版本等信息未进行说明。

（6）测试数据的要求不详细。测试数据的准备情况影响测试的进度和效率，因此需要对测试数据的要求尽早规划，以便从用户、研制人员等处获得测试数据，保证测试的顺利实施。

（7）测试环境差异性分析的不充分。测试环境直接影响测试结果，特别是性能等测试，应进行充分地分析，以便保证测试结果的可信性。

（8）测试结束条件和测试通过准则不具体，可操作性不强。测试结束条件和测试通过准则需要与委托方进行充分沟通，获得具体、可操作、可实施的测试结束条件和测试通过准则。

## 12.3　测试设计与实现

测试设计与实现是测试过程中关键的环节之一。调查显示，有 84%的被调查者认为，测试设计与实现工作是对软件测试质量影响最大的工作。因此，做好测试设计与实现工作，对保障测试的充分性至关重要。

### 12.3.1 测试设计与实现的内容

测试设计与实现的主要工作是依据软件开发文档和测试计划，进行测试用例的设计，并完成测试所需的测试环境的建立和验证。测试设计与实现阶段完成时，应编写软件测试说明、测试环境验证情况报告。测试设计与实现的内容包括以下内容。

（1）对软件测试计划中定义的所有测试项设计测试用例，测试用例说明的要求如下。

① 说明测试用例名称和标识。

② 建立测试用例的追踪。说明测试用例所测试内容的追踪关系，例如追踪到软件需求规格说明中的相关需求的标识。

③ 说明测试用例综述。简要描述测试目的和所采用的测试方法。

④ 提出测试用例的初始化要求。测试用例初始化要求包括硬件配置、软件配置（包括测试的初始条件）、测试配置（例如用于测试的模拟系统和测试工具）、参数设置（例如测试开始前对初始化数据的设置）等的初始化要求。

⑤ 定义测试用例的输入。每个测试用例输入的描述包括：每个测试输入的名称和具体内容，例如确定的数值、状态或信号等；测试输入的来源，例如测试程序产生信息、磁盘文件、通过网络接收信息、人工键盘输入等；测试输入是真实的还是模拟的；测试输入的时间顺序或事件顺序。

⑥ 说明测试用例的期望测试结果。期望测试结果应有具体内容，例如确定的数值、状态或信号等，不应是不确切的概念或笼统的描述。必要时，应提供中间的期望结果。

⑦ 说明测试用例的测试结果评估准则。评估准则用以判断测试用例执行中产生的中间或最后结果是否正确。评估准则应根据不同情况提供相关信息，例如：实际测试结果所需的精确度；允许的实际测试结果与期望结果之间差异的上、下限；时间的最大或最小间隔；事件数目的最大或最小值；实际测试结果不确定时，重新测试的条件；与产生测试结果有关的出错处理。

⑧ 设计测试用例的执行步骤。编写按照执行顺序排列的一系列相对独立的步骤，执行步骤应包括：每一步所需的测试操作动作、测试输入或设备操作等；每一步期望的测试结果；每一步的评估准则；导致被测程序执行终止伴随的动作或指示信息；需要时，获取和分析中间结果的办法。

⑨ 说明测试用例的前提和约束。在测试用例中应说明实施测试用例的前提条件和约束条件，例如特别限制、参数偏差或异常处理等，并要说明它们对测试用例的影响。

⑩ 规定测试终止条件。说明测试用例的测试正常终止和异常终止的条件。

⑪ 确定测试说明与测试计划的追踪关系，给出清晰、明确的追踪表。

⑫ 编写软件测试说明。

（2）根据测试资源、风险等约束条件确定测试用例执行顺序。

（3）针对测试输入要求，设计测试数据，准备和验证所有的测试数据。

（4）准备并获取测试资源，例如测试工具、搭建测试环境所必需的软硬件资源等。

（5）必要时，开发测试执行所需程序，例如开发单元测试、集成测试的驱动模块、桩模块以及测试支持软件等。

（6）建立和校核测试环境，记录校核结果，分析拟建立的测试环境与需求环境之间的差异；如果存在环境差异，应说明在该测试环境下测试结果的有效性。

## 12.3.2 测试说明模板

1 范围
1.1 标识
1.2 被测软件概述
1.3 文档概述
2 引用文档
3 术语和定义
4 测试准备
4.X （被测对象）
4.X.1 测试进度
4.X.2 硬件准备
在本次测试中对某个被测对象进行测试要用到的硬件。
4.X.3 软件准备
在本次测试中对某个被测对象进行测试要用到的软件。
4.X.4 其他测试准备
描述为完成测试所必需的任何其他测试准备工作或过程。
5 测试说明
5.X （被测对象）
5.X.Y （被测对象的测试类型）
5.X.Y.Z （被测对象的测试项）
5.X.Y.Z.K （被测对象的测试用例）

<table>
<tr><td>测试用例名称</td><td colspan="2"></td><td colspan="2">用例标识</td><td></td></tr>
<tr><td>追踪关系</td><td colspan="2"></td><td colspan="2"></td><td></td></tr>
<tr><td>测试用例综述</td><td colspan="2"></td><td colspan="2"></td><td></td></tr>
<tr><td>用例初始化</td><td colspan="2"></td><td colspan="2"></td><td></td></tr>
<tr><td>前提和约束</td><td colspan="2"></td><td colspan="2"></td><td></td></tr>
<tr><td>设计方法</td><td colspan="2"></td><td colspan="2"></td><td></td></tr>
<tr><td>测试步骤</td><td colspan="2"></td><td colspan="2"></td><td></td></tr>
<tr><td>序号</td><td colspan="3">输入及操作</td><td colspan="2">期望结果与评估标准</td></tr>
<tr><td></td><td colspan="3"></td><td colspan="2"></td></tr>
<tr><td colspan="2">测试用例终止条件</td><td colspan="4"></td></tr>
<tr><td colspan="2">测试用例通过准则</td><td colspan="4"></td></tr>
</table>

6 测试用例追踪关系

表 6-1 测试项与测试用例的追踪关系表

| 序号 | 测试项标识 | 测试名称 | 测试用例标识 | 测试用例名称 |
|---|---|---|---|---|
| | | | | |

表 6-2 测试用例与测试项的追踪关系表

| 序号 | 测试用例标识 | 测试用例名称 | 测试项标识 | 测试项名称 |
|---|---|---|---|---|
| | | | | |

## 12.3.3 测试设计与实现常见问题

测试设计与实现是软件测试关键环节，测试用例设计得是否合理、有效，将直接影响测试的充分性，构建测试环境的符合性也直接影响测试结果的有效性，因此对该工作需要高度重视。软件测试设计与实现常见如下问题。

（1）测试环境描述的软/硬件资源与测试计划中提出的测试环境要求不一致。

（2）缺少数据准备的说明。测试数据是实施测试的必备条件，应充分考虑各类测试数据的准备情况，因此需要在测试说明中较为详细地说明测试数据的准备情况。

（3）测试用例的追踪关系不正确。

（4）缺少用例初始化、前提和约束条件的说明。用例初始化、前提和约束条件是测试用例执行的基础，应具体说明。

（5）未采取恰当的测试用例设计方法。设计方法是保证测试用例设计合理性、充分性的重要手段，测试用例设计时需要根据需要合理地应用必要的测试用例设计方法。

（6）测试用例设计时，未按照测试项中提出的测试用例设计要求进行相应设计。另外，测试用例描述的测试方法与测试项中要求的测试方法不一致。这个问题发生的主要原因可能是由于测试用例设计时，发现测试项中提出的测试方法不恰当，但未及时更新测试计划中测试项的测试方法的说明而造成的。

（7）测试输入及操作描述不具体。测试用例执行时，需要根据具体的测试输入及操作进行，因此应具体、明确地进行说明。但是在实施时，经常出现测试输入及操作模糊、不明确和不具体的问题。

（8）期望结果是描述预期的期望结果，需要根据期望结果判断测试是否通过的重要依据，但经常发生期望结果不明确、不具体的问题。

（9）评估标准是判断测试步骤是否通过的依据，但在实际工作中往往出现期望结果描述较为笼统，甚至出现未说明如何判断正确与否的情况。

（10）对于各种测试类型下设计的测试用例，其具体设计各有特点，常出现下列问题：

① 文档审查中审查内容不具体、明确，只有简单的“文档一致性”审查条款，应根据每个文档内容的不同特点制定较为具体、可操作的检查内容，另外应注意文档检查内容与开发文档应遵循的文档规范是否一致，并得到相关方的认可，特别是应得到委托方的同意；

② 代码审查不完整，代码审查中往往考虑了用工具进行代码执行标准情况的审查和代码可读性审查，却忽略了较为关键的代码和设计的一致性检查和代码逻辑表达的正确性审查；

③ 静态分析不全面，静态测试可以利用工具进行辅助分析，但是工具辅助检查的结果需要进行进一步的分析，使人工分析的工作量较大，这是测试中往往容易忽略的内容；

④ 代码走查未说明选择关键代码的原则，代码走查一般是在代码审查、静态分析的基础上，对问题较多，比较关键的模块或测试无法覆盖的部分进行的测试类型，需要根据软件代码设计用例，通过人工模拟计算机运行，检查输出结果是否正确，常见问题主要表现在未说明走查内容选取的原则、用例的输入和输出结果不明确等；

⑤ 逻辑测未分析测试未覆盖的原因，逻辑测试常见问题主要表现在对未达到测试覆盖要求的部分未进行分析，未对未覆盖部分补充相应的测试用例；

⑥ 功能测试不全面，功能测试时，容易忽略对非法边界输入的测试，未考虑超负荷、饱和情况和其他“最坏情况”的测试，在配置项测试时，缺少对控制流程的正确性、合理性地测试；

⑦ 性能测试未考虑环境因素，性能测试时，未考虑软件运行环境可能对性能指标的影响，另外在性能指标的度量时，未采用比指标要求精度更高的测量方法进行测量；

⑧ 接口测试时对接口异常情况的考虑不充分，对数据内容本身的错误考虑欠缺；

⑨ 在人机交互界面的测试中未对用户手册的一致性进行检查，对错误操作流程的测试考虑不充分；

⑩ 强度测试未考虑在软件达到饱和指标后，再恢复到正常状态的测试过程；

⑪ 余量测试时，只考虑资源的余量，未考虑功能处理时间的余量；

⑫ 安全性测试时，只对保密安全性进行了测试，对防止危险状态措施有效性的测试内容考虑不全面；

⑬ 边界测试时，只考虑输入边界测试，对输出边界的测试不充分；

⑭ 安装测试时，只进行安装测试，未对卸载进行测试，也未对卸载后重新安装的正确性进行测试；

⑮ 兼容性测试时，只考虑了新版本与旧版本之间的兼容性，对与其他软件的兼容性考虑不充分。

（11）测试用例终止条件描述得不恰当，常见问题是描述得都一样，例如“本测试用例的全部测试步骤被执行或因某种原因导致测试步骤无法执行（异常终止）”。应根据不同测试用例的具体情况说明测试终止条件。

（12）测试用例通过准则描述的不具体，常见问题是所有测试用例的通过准则都描述得一样，例如“本测试用例的全部测试步骤都通过即标志本用例为‘通过’”。应根据不同测试用例的具体情况说明测试用例通过准则。

（13）测试设计与实现另一个重要活动就是准备测试环境和数据，并要求对测试环境和测试数据进行验证。常见问题是测试环境和测试数据的验证不充分，导致测试实施时，需要大量时间调整测试环境和测试数据，有时因为测试环境和测试数据的问题影响部分测试用例的执行。

（14）测试用例执行顺序的制定，需要根据测试资源、测试优先级等因素确定。常见问

题是未制定测试顺序，现场测试临时确定测试顺序，造成部分测试资源的浪费，或与其他测试工作发生冲突等问题。

## 12.4 测试执行

测试执行是发现问题的关键环节，通过在规定的测试环境下，执行设计的测试用例，获得测试结果，对测试结果进行分析，提出软件问题报告，最终解决软件的问题。

### 12.4.1 测试执行的内容

测试执行是依据软件测试计划、测试说明，按测试用例执行顺序进行测试，记录实际的测试结果，当发现问题时应进行分析。如果是测试用例、环境、数据等的问题，应根据实际情况调整测试计划和说明，补充相应的测试；如果是软件的问题，应如实、详细地记录测试结果，并提交问题报告。问题报告中应详细描述问题现象，分析问题类别和严重性等级，给出改进意见建议，为后续的问题定位、解决提供支持。

如果软件进行了更改，应进行回归测试。回归测试的要求见第8章的内容，回归测试应编写回归测试方案。执行回归测试时，应记录回归测试记录，如果仍存在问题应填写问题报告。

### 12.4.2 测试执行模板

测试记录
1 范围
1.1 标识
1.2 被测软件概述
1.3 文档概述
1.4 与其他文档的关系
2 引用文档
3 术语和定义
4 测试环境验证
4.1 硬件环境
在本次测试中对某个被测对象进行测试要用到的硬件验证情况。
4.2 软件环境
在本次测试中对某个被测对象进行测试要用到的软件验证情况。
4.3 测试数据
描述为完成测试所必需的测试数据验证情况。
4.4 测试环境验证记录

5 测试记录

5.X （被测对象）

5.X.Y （被测对象的测试类型）

5.X.Y.Z （被测对象的测试项）

5.X.Y.Z.K （被测对象的测试用例）

| | | | |
|---|---|---|---|
| 测试用例名称 | | 标 识 | |
| 追踪关系 | | | |
| 测试用例综述 | | | |
| 用例初始化 | | | |
| 前提和约束 | | | |
| 设计方法 | | | |
| 测试步骤 | | | |
| 序号 | 输入及操作 | 期望结果与评估标准 | 实测结果 |
| | | | |
| 测试用例终止条件 | | | |
| 测试用例通过准则 | | | |
| 执行状态 | | 执行结果 | |
| 问题单标识 | | | |
| 测试人员 | | | |
| 测试时间 | | | |

6 问题报告

| | | | | | |
|---|---|---|---|---|---|
| 问题标识 | | | | | |
| 被测对象 | | | | | |
| 问题类别 | □程序错误 | □文档错误 | □设计错误 | 其他错误 | |
| 问题严重性等级 | □1 级 | □2 级 | □3 级 | □4 级 | |
| 关联的测试用例及其在测试记录中的章节号 | | | | | |
| | | | | | |
| 问题描述： | | | | | |
| 附注及修改建议： | | | | | |
| 报告人 | | 报告日期 | | | |
| 设计师意见： | | | | | |
| 测试人员意见： | | | | | |
| 总师意见 | | | | | |

软件回归测试方案
1 范围
1.1 标识
1.2 被测软件概述
1.3 文档概述
1.4 与其他文档的关系
2 引用文档
3 术语和定义
4 测试环境
4.1 测试环境概述
4.2 测试环境配置
4.3 测试环境的安装、验证与控制
4.3.1 安装
4.3.2 验证
4.3.3 控制
4.3.4 测试环境差异与有效性分析
5 测试人员安排
6 测试进度计划
7 测试策略
8 测试分析与设计
8.1 软件更动说明
8.1.1 问题引发更动说明
8.1.2 其他软件更动说明
8.2 软件更动影响域分析
9 回归测试说明
9.1 （测试类型名称）
9.1.1 （测试项名称）
9.1.1.1 （测试用例名称）
10 测试追踪关系

### 12.4.3 测试实施常见问题

测试实施是最终发现软件问题，提供软件改进意见的关键环节，其客观性、准确性是问题定位和解决的依据。因此，应对测试实施过程中活动进行控制，保证其客观性和准确性。测试实施中常见如下问题。

（1）未按照测试执行顺序执行导致测试实施混乱。

（2）未依照测试用例设计执行，导致测试内容的遗漏。测试过程中应允许测试人员根据测试的执行情况调整部分的测试用例。但是，应在对现有用例执行情况分析的基础上

进行。

(3) 测试记录不客观、不准确。常见测试结果记录为“通过”,这是对测试结果的判断,而不是实测结果。测试过程中应客观、准确地记录实测结果,并在此基础上判断测试结果。特别是对于有量值的实测结果,应准确、如实记录。

(4) 问题描述不详细。常出现问题描述只有测试人员自己了解,开发人员无法依据此进行准确分析和定位问题的情况。

(5) 问题严重性等级定义较为随意。应根据准确、统一的标准定义软件问题严重性,为软件问题的解决提供决策。

(6) 回归测试方案中,测试策略应说明回归测试的策略而不是整个测试活动的测试策略。

(7) 更动分析应根据研制人员提供的更动分析,再进行进一步的分析,以确定回归测试的范围。常见问题表现在未进行分析或分析得不充分。对问题不更动时,没有得到相关人员的同意。

## 12.5 测试总结

通过测试总结对测试工作的情况进行分析,汇总及判断测试工作的完成情况和软件的质量情况,以便确定测试是否终止。

### 12.5.1 测试总结的内容

测试完成后应对测试工作进行总结,以便评估软件测试中的问题是否得到解决、测试工作是否满足充分性要求。测试结果分析总结应写入软件测试报告,并应进行评审。测试总结的内容包括以下内容。

(1) 对测试过程进行总结。应对测试策划、测试设计与实现、测试执行过程进行总结,说明在上述各过程中开展的主要工作、参与人员和工作完成情况。

(2) 对测试方法进行说明。应说明测试所采用的测试方法与策略,并说明采用这些方法的依据。

(3) 对测试环境进行分析。应说明测试所使用的测试环境,包括测试工具、桩模块和驱动模块的情况,并对测试环境的差异性进行分析,说明测试环境是否满足测试的要求。

(4) 对测试结果进行分析。测试结果的分析应包括对测试执行过程以及所有回归测试的情况的分析。主要内容包括:

① 测试时间;

② 测试人员;

③ 测试用例执行情况,包括测试用例数、通过的测试用例、未通过的测试用例、完全执行的测试用例、未完全执行的测试用例、未执行的测试用例;

④ 测试覆盖情况,包括对功能、性能、接口等覆盖情况,说明是否满足测试充分性要求;

⑤ 说明测试过程中发现的问题,并对问题解决情况进行说明;

⑥ 对软件的整体情况进行评价,并提出改进的意见及建议。

## 12.5.2 测试总结模板

1 范围
1.1 标识
1.2 被测软件概述
1.3 文档概述
1.4 与其他文档的关系
2 引用文档
3 术语和定义
4 测试概述
4.1 测试过程概述
4.1.1 测试策划
4.1.2 测试设计和实现
4.1.3 测试执行
4.2 测试环境说明
4.2.1 软硬件环境
4.2.2 测试场所
4.2.3 测试数据
4.2.4 环境差异影响分析
4.3 测试方法说明 5 测试结果
5 测试情况
5.1 执行测试情况
5.1.1 首次测试
说明首次测试的被测软件版本、测试时间、测试人员和用例执行情况。
5.1.X 第X次回归测试
本次测试的被测对象版本为：X.XX
5.1.1.1 测试时间
测试开始日期：20XX-XX-XX
测试结束日期：20XX-XX-XX
5.1.1.2 测试人员
5.1.1.3 测试用例执行情况

**表 5-1 测试用例执行情况统计**

| | | | |
|---|---|---|---|
| 测试项个数 | | | |
| 设计的测试用例总数 | | | |
| 完全执行的测试用例数 | | 通过的测试用例数 | |
| | | 未通过的测试用例数 | |
| 部分执行的测试用例数 | | 未通过的测试用例数 | |
| 未执行的测试用例数 | | | |

表 5-2　测试依据与测试项追踪关系表

| 序号 | 测试依据标识 | 测试依据 | 测试项名称或未追踪原因说明 |
| --- | --- | --- | --- |
| | | | |

表 5-3　××××软件评测下属测试类、测试项、测试用例关系表

| 序号 | 测试类名称/标识 | 测试项名称/标识 | 测试用例名称/标识 |
| --- | --- | --- | --- |
| | | | |

表 5-4　××××软件评测下属测试用例执行结果统计表

| 序号 | 测试用例名称 | 测试用例标识 | 执行状态 | 执行结果 | 问题步骤 | 问题报告单标识 |
| --- | --- | --- | --- | --- | --- | --- |
| | | | | | | |

5.1.1.4　未完整执行测试用例的原因说明

对未完整执行的测试用例进行原因说明。

5.1.1.5　测试执行情况的其他说明

无。

5.2　软件问题

5.2.X　第 X 次测试

测试问题详细说明见软件问题报告。

表 5-5　××××软件测试提交问题分类统计表

| 问题级别 / 问题类别 | 第 1 级问题 | 第 2 级问题 | 第 3 级问题 | 第 4 级问题 | 第 5 级问题 |
| --- | --- | --- | --- | --- | --- |
| 程序问题 | | | | | |
| 文档问题 | | | | | |
| 设计问题 | | | | | |
| 其他问题 | | | | | |
| 总计 | | | | | |

表 5-6　××××软件测试提交问题在测试类型中分布统计表

| 问题严重性等级 / 测试类型 | 第 1 级问题 | 第 2 级问题 | 第 3 级问题 | 第 4 级问题 | 第 5 级问题 |
| --- | --- | --- | --- | --- | --- |
| 安装性测试 | | | | | |
| 文档审查 | | | | | |
| 功能性测试 | | | | | |
| 接口测试 | | | | | |

续表

| 问题严重性等级<br>测试类型 | 第 1 级问题 | 第 2 级问题 | 第 3 级问题 | 第 4 级问题 | 第 5 级问题 |
|---|---|---|---|---|---|
| 人机交互界面测试 | | | | | |
| 可靠性测试 | | | | | |
| 安全性测试 | | | | | |
| 余量测试 | | | | | |
| 强度测试 | | | | | |
| …… | | | | | |
| 总计 | | | | | |

**表 5-7 软件问题报告处置情况及其影响域分析一览表**

| 序号 | 软件问题标识 | 是否已更动 | 影响域分析 | |
|---|---|---|---|---|
| | | | 说明 | 涉及的测试依据 |
| | | | | |

5.3 测试的有效性、充分性说明

对照测试计划，根据测试需求分析、测试策划、测试设计与实现、测试执行、测试总结等阶段的实施情况以及发现的软件问题，对测试的有效性、充分性进行分析说明。

6 评价结论与改进建议

6.1 评价结论

对照测试计划中的相关规定，对照被测软件研制要求或系统研制要求、软件需求规格说明及其他等效文档规定的书面要求和隐含要求，结合测试结果，对被测软件的质量作全面评价，对被测软件满足研制总要求的情况，是否通过测试给出明确的结论。

6.2 改进建议

结合测试的具体情况，提出对被测软件质量的改进建议。

### 12.5.3 测试总结常见问题

测试总结不仅对被测软件的情况进行分析，为软件质量的改进提供依据，还需要对测试工作本身的情况进行总结，为测试工作的持续改进提供依据。测试总结常见如下问题。

(1) 被测软件的描述不全面，经常缺少软件关键等级、使用环境、主要技术指标、软件规模与开发语言等信息。

(2) 在说明测试过程时，当存在有多轮测试的情况时，应说明每次测试的软件版本，以及软件主要的更改情况。对测试用例执行情况、问题情况的分析和描述要准确。经常出现缺少软件版本的描述，测试用例执行情况、问题情况的描述与实际不一致的问题。

(3) 测试环境的描述与测试计划和测试说明中的描述不一致，对与软件实际运行环境

中存在差异的情况未进行说明，也未分析在此环境下实施的测试所获得测试结果的可信性。

（4）对测试用例执行情况分析不准确、不全面。对测试用例执行数据分析不准确，对未执行或部分执行的测试用例未说明具体原因。

（5）对软件问题的统计分析数据不准确。对按问题严重性等级/问题类别、测试类型/问题严重性等级的数据分析不准确。

（6）对未修改的问题没有进行详细、深入地分析，未说明可能对系统造成的影响。

（7）对软件的评价客观性不足。主要表现在评价结论笼统，使用了不确定的用语，例如"基本满足"。建议用表格化方式列出软件应满足的功能、性能、接口等要求，并明确说明每一项的满足情况。

（8）缺少对被测软件的改进意见。应根据软件测试的情况对被测软件质量、管理等方面提出改进意见和建议。

# 参 考 文 献

[1] SPILLNER A,等. 软件测试基础教程[M]. 2 版. 刘琴,等,译. 北京：人民邮电出版社,2009.

[2] LEWIS W E,VEERAPILLAI G. 软件测试与持续质量改进[M]2 版. 陈绍英,等,译. 北京：人民邮电出版社,2009.

[3] 古乐,等. 软件测试技术概论[M]. 北京：清华大学出版社,2004.

[4] 中华人民共和国国家质量监督检验检疫总局,中国国家标准化管理委员会. 信息技术　软件工程术语：GB/T 11457—2006[S]. 北京：中国标准出版社,2006.

[5] 朱少民,等. 软件测试方法和技术[M]. 北京：清华大学出版社,2005.

[6] 郁莲. 软件测试方法与实践[M]. 北京：清华大学出版社,2008.

[7] 武剑洁. 软件测试使用教程——方法与实践[M]. 2 版. 北京：电子工业出版社,2012.

[8] Jones C. Applied Software Measurement：Global Analysis of Productivity and Quality,3rd Edition [M]. New York：McGraw-Hill,2008.

[9] Mustafa K,Khan R A. 软件测试：概念与实践[M]. 董威,译. 北京：科学出版社,2009.

[10] 中华人民共和国国家质量监督检验检疫总局,中国国家标准化管理委员会. 计算机软件测试规范：GB/T 15532—2008[S]. 北京：中国标准出版社,2008.

[11] 张卫祥,刘文红. 一种基于组合覆盖的黑盒测试用例自动生成方法[J]. 飞行器测控学报,2008,27(5)：53-56.

[12] 蔡建平. 软件测试大学教程[M]. 北京：清华大学出版社,2009.

[13] 中华人民共和国国家质量监督检验检疫总局,中国国家标准化管理委员. 系统与软件可靠性　第 3 部分：测试方法：GB/T 29832. 3—2013[S]. 北京：中国标准出版社,2013.

[14] 李必信. 程序切片技术及其应用[M]. 北京：科学出版社,2006：170-173.

[15] WANG X. Application of UML Statechart Diagram in Regression Test[J]. Computer Engineering, 2009,5(4)：63-65.

[16] CHEN SF, ZHENG HY. Dependence analysis and regression testing of object-oriented software [J]. Journal of Computer Applications. 2009, 29 (11)：3110-3113.

[17] AMMANN P, OFFUTT J. Introduction to Software Testing [M]. Cambridge, Eng：Cambridge University Press,2008.

[18] 聂长海. 软件测试的概念与方法[M]. 北京：清华大学出版社,2013.

[19] HARROLD M J. Testing：a roadmap [J]. ICSE-Future of SE Track, 2000：61-72.

[20] BERTOLINO A. Software Testing Research：Achievements, Challenges, Dreams[J]. Future of Software Engineering,2007：85-103.

[21] 单锦辉，姜瑛，孙萍. 软件测试研究进展[J]. 北京大学学报(自然科学版), 2005, 41(1)：134-145.

[22] VOAS J, MORELL L, MILLER K. Predicting where faults can hide from testing [J]. IEEE Software,1999,3：41-48.

[23] 中国人民解放军总装备部. 军用软件开发通用要求：GJB 2786A—2009[S]. 北京：中国标准出版社,2009.

[24] 黄隶凡,郑学仁. FPGA 设计中的亚稳态研究[J]. 微电子学,2011,41(2)：265-268.

[25] 吴小蕻. 跨时钟域若干问题的研究——同步与亚稳态[D]. 合肥：中国科技大学,2008.

[26] 赵永建,段国东,李苗. 集成电路中的多时钟域同步设计技术[J]. 计算机工程,2008,34(14)：

246-247.

[27] 樊民革，赵剡. 一种消除异步电路亚稳态的逻辑控制方法[J]. 电子测量技术. 2008，31(10)：24-27.

[28] 汪路元. FPGA 设计中的亚稳态及其缓解措施[J]. 电子技术应用，2012，38(8)：13-19.

[29] 万敏. 异步时序电路中的亚稳态设计与分析[D]. 上海：上海交通大学，2008.

[30] 蒲石. 异步多时钟域系统的同步设计研究[D]. 西安：西安电子科技大学，2007.

[31] 廖永波. SOC 软硬件协同方法及其在 FPGA 芯片测试中的应用研究[D]. 西安：西安电子科技大学，2010.

[32] 王春平，张晓华，赵翔. Xilinx 可编程逻辑器件设计与开发(基础篇)[M]. 北京：人民邮电出版社，2011.